Structure

1.0 Objectives

At the end of this chapter the reader will be able to:

- Distinguish between data and information and Knowledge
- Distinguish between file processing system and DBMS
- Describe DBMS its advantages and disadvantages
- Describe Database users including data base administrator
- Describe data models, schemas and instances.
- Describe DBMS Architecture & Data Independence
- Describe Data Languages

1.1 Introduction

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data. This is a collection of related data with an implicit meaning and hence is a database. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*. By **data,** we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. You may have recorded this data in an indexed address book, or you may have stored it on a diskette, using a personal computer and software such as DBASE IV or V, Microsoft ACCESS, or EXCEL. A **datum** – a unit of data – is a symbol or a set of symbols which is used to represent something. This relationship between symbols and what they represent is the essence of what we mean by **information**. Hence, information is interpreted data – data supplied with semantics. **Knowledge** refers to the practical use of information. While information can be transported, stored or shared without many difficulties the same can not be said about knowledge. Knowledge necessarily involves a personal experience. Referring back to the scientific experiment, a third person reading the results will have information about it, while the person who conducted the experiment personally will have knowledge about it.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Because information is so important in most organizations, computer scientists have developed a large body of concepts and techniques for managing data. These concepts and technique form the focus of this book. This chapter briefly introduces the principles of database systems.

1.2 Data Processing Vs. Data Management Systems

Although Data Processing and Data Management Systems both refer to functions that take raw data and transform it into usable information, the usage of the terms is very different. **Data Processing** is the term generally used to describe what was done by large mainframe computers from the late 1940's until the early 1980's (and which continues to be done in most large organizations to a greater or lesser extent even today): large volumes of raw transaction data fed into programs that update a master file, with fixed-format reports written to paper.

The term **Data Management Systems** refers to an expansion of this concept, where the raw data, previously copied manually from paper to punched cards, and later into data-entry terminals, is now fed into the system from a variety of sources, including ATMs, EFT, and direct customer entry through the Internet. The master file concept has been largely displaced by database management systems, and static reporting replaced or augmented by ad-hoc reporting and direct inquiry, including downloading of data by customers. The ubiquity of the Internet and the Personal Computer have been the driving force in the transformation of Data Processing to the more global concept of Data Management Systems.

1.3 File Oriented Approach

The earliest business computer systems were used to process business records and produce information. They were generally faster and more accurate than equivalent manual systems. These systems stored groups of records in separate files, and so they were called **file processing systems.** In a typical file processing systems, each department has its own files, designed specifically for those applications. The department itself working with the data processing staff, sets policies or standards for the format and maintenance of its files.

Programs are dependent on the files and vice-versa; that is, when the physical format of the file is changed, the program has also to be changed. Although the traditional file oriented approach to information processing is still widely used, it does have some very important disadvantages.

1.4 Database Oriented Approach to Data Management

Consider part of a savings-bank enterprise that keeps information about all customers and savings accounts. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including

A program to debit or credit an account

A program to add a new account

A program to find the balance of an account

A program to generate monthly statements

System programmers wrote these application programs to meet the needs of the bank. New application programs are added to the system as the need arises. For example, suppose that the savings bank decides to offer checking accounts. As a result, the bank creates new permanent files that contain information about all the checking accounts maintained in the bank, and it may have to write new application programs to deal with situations that do not arise in savings accounts, such as overdrafts. Thus, as time goes by, the system acquires more files and more application programs.

This typical **file-processing system** is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMSs) came along, organizations usually stored information in such systems.

Keeping organizational information in a file-processing system has a number of major disadvantages:

Data redundancy and inconsistency.

Since different programmers create the files and application programs over a long period, the various files are likely to have different formats and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, the address and telephone number of a particular customer may appear in a file that consists of savings-account records and in a file that consists of checking-account records. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies

of the same data may no longer agree. For example, a changed customer address may be reflected in savings-account records but not elsewhere in the system.

Difficulty in accessing data.

Suppose that one of the bank officers needs to find out the names of all customers who live within a particular postal-code area. The officer asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of *all* customers. The bank officer has now two choices: either obtain the list of all customers and extract the needed information manually or ask a system programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same officer needs to trim that list to include only those customers who have an account balance of $10,000 or more. As expected, a program to generate such a list does not exist. Again, the officer has the preceding two options, neither of which is satisfactory.

The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

Data isolation. Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

Integrity problems. The data values stored in the database must satisfy certain types of **consistency constraints**. For example, the balance of a bank account may never fall below a prescribed amount (say, $25). Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

Atomicity problems. A computer system, like any other mechanical or electrical device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure. Consider a program to transfer $50 from account *A* to account *B*. If a system failure occurs during the execution of the program, it is possible that the $50 was removed from account *A* but was

not credited to account *B*, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

Concurrent-access anomalies. For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates may result in inconsistent data. Consider bank account *A*, containing $500. If two customers withdraw funds (say $50 and $100 respectively) from account *A* at about the same time, the result of the concurrent executions may leave the account in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $500, and write back $450 and $400, respectively. Depending on which one writes the value last, the account may contain either $450 or $400, rather than the correct value of $350. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.

Security problems. Not every user of the database system should be able to access all the data. For example, in a banking system, payroll personnel need to see only that part of the database that has information about the various bank employees. They do not need access to information about customer accounts. But, since application programs are added to the system in an ad hoc manner, enforcing such security constraints is difficult.

These difficulties, among others, prompted the development of database systems. In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems. In most of this book, we use a bank enterprise as a running example of a typical data-processing application found in a corporation.

1.5 Characteristics of Database

The database approach has some very characteristic features which are discussed in detail below:

1.5.1 Concurrent Use

A database system allows several users to access the database concurrently. Answering different questions from different users with the same (base) data is a central aspect of an information system. Such concurrent use of data increases the economy of a system.

An example for concurrent use is the travel database of a bigger travel agency. The employees of different branches can access the database concurrently and book journeys for their clients. Each travel agent sees on his interface if there are still seats available for a specific journey or if it is already fully booked.

1.5.2 Structured and Described Data

A fundamental feature of the database approach is that the database systems does not only contain the data but also the complete definition and description of these data. These descriptions are basically details about the extent, the structure, the type and the format of all data and, additionally, the relationship between the data. This kind of stored data is called metadata ("data about data").

1.5.3 Separation of Data and Applications

As described in the feature structured data the structure of a database is described through *metadata* which is also stored in the database. An application software does not need any knowledge about the physical data storage like encoding, format, storage place, etc. It only communicates with the management system f a database (DBMS) via a standardised interface with the help of a standardised language like SQL. The access to the data and the metadata is entirely done by the DBMS. In this way all the applications can be totally seperated from the data. Therefore database internal reorganisations or improvement of efficiency do not have any influence on the application software.

1.5.4 Data Integrity

Data integrity is a byword for the quality and the reliability of the data of a database system. In a broader sense data integrity includes also the protection of the database from unauthorised access (confidentiality) and unauthorised changes. Data reflect facts of the real world. database.

1.5.5 Transactions

A transaction is a bundle of actions which are done within a database to bring it from one

consistent state to a new consistent state. In between the data are inevitable inconsistent. A transaction is atomic what means that it cannot be divided up any further. Within a transaction all or none of the actions need to be carried out. Doing only a part of the actions would lead to an inconsistent database state. One example of a transaction is the transfer of an amount of money from one bank account to another. The debit of the money from one account and the credit of it to another account makes together a consistent transaction. This transaction is also atomic. The debit or credit alone would both lead to an inconsistent state. After finishing the transaction (debit and credit) the changes to both accounts become persistent and the one who gave the money has now less money on his account while the receiver has now a higher balance.

1.5.6 Data Persistence

Data persistence means that in a DBMS all data is maintained as long as it is not deleted explicitly. The life span of data needs to be determined directly or indirectly be the user and must not be dependent on system features. Additionally data once stored in a database must not be lost. Changes of a database which are done by a transaction are persistent. When a transaction is finished even a system crash cannot put the data in danger.

1.6 Advantages and Disadvantages of a DBMS

Using a DBMS to manage data has many advantages:

Data independence: Application programs should be as independent as possible from details of data representation and storage. The DBMS can provide an abstract view of the data to insulate application code from such details.

Efficient data access: A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.

Data integrity and security: If data is always accessed through the DBMS, the DBMS can enforce integrity constraints on the data. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, the DBMS can enforce *access controls* that govern what data is visible to different classes of users.

Data administration: When several users share the data, centralizing the administration of data can offer significant improvements. Experienced professionals who understand the nature of the data being managed, and how different groups of users use it, can be responsible for organizing the data representation to minimize redundancy and fine-tuning the storage of the data to make retrieval efficient.

Concurrent access and crash recovery: A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.

Reduced application development time: Clearly, the DBMS supports many important functions that are common to many applications accessing data stored in the DBMS. This, in conjunction with the high-level interface to the data, facilitates quick development of applications. Such applications are also likely to be more robust than applications developed from scratch because many important

tasks are handled by the DBMS instead of being implemented by the application. Given all these advantages, is there ever a reason *not* to use a DBMS? A DBMS is a complex piece of software, optimized for certain kinds of workloads (e.g., answering complex queries or handling many concurrent requests), and its performance may not be adequate for certain specialized applications. Examples include applications with tight real-time constraints or applications with just a few well-designed critical operations for which efficient custom code must be written. Another reason for not using a DBMS is that an application may need to manipulate the data in ways not supported by the query language. In such a situation, the abstract view of the data presented by the DBMS does not match the application's needs, and actually gets in the way. As an example, relational databases do not support flexible analysis of text data (although vendors are now extending their products in this direction). If specialized performance or data manipulation requirements are central to an application, the application may choose not to use a DBMS, especially if the added benefits of a DBMS (e.g., flexible querying, security, concurrent access, and crash recovery) are not required. In most situations calling for large-scale data management, however, DBMSs have become an indispensable tool.

Disadvantages of a DBMS

Danger of a Overkill: For small and simple applications for single users a database system is often not advisable.

Complexity: A database system creates additional complexity and requirements. The supply and operation of a database management system with several users and databases is quite costly and demanding.

Qualified Personnel: The professional operation of a database system requires appropriately trained staff. Without a qualified database administrator nothing will work for long.

Costs: Through the use of a database system new costs are generated for the system itselfs but also for additional hardware and the more complex handling of the system.

Lower Efficiency: A database system is a multi-use software which is often less efficient than specialised software which is produced and optimised exactly for one problem.

1.7 Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all.

The concept of database schemas and instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program. Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema.

Database systems have several schemas, partitioned according to the levels of abstraction.

The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level.Adatabase may also have several schemas at the view level, sometimes called **subschemas**, that describe different views of the database.

Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

We study languages for describing schemas, after introducing the notion of data models in the next section.

1.8 Data Models

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

To illustrate the concept of a data model, we outline two data models in this section: the entity-relationship model and the relational model. Both provide a way to describe the design of a database at the logical level.

1.8.1 The Entity-Relationship Model

The entity-relationship (E-R) data model is based on a perception of a real world that consists of a collection of basic objects, called *entities*, and of *relationships* among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. For example, each person is an entity, and bank accounts can be considered as entities.

Entities are described in a database by a set of **attributes**. For example, the attributes *account-number* and *balance* may describe one particular account in a bank, and they form attributes of the *account* entity set. Similarly, attributes *customer-name*, *customer-street* address and *customer-city* may describe a *customer* entity.

An extra attribute *customer-id* is used to uniquely identify customers (since it may be possible to have two customers with the same name, street address, and city).

A unique customer identifier must be assigned to each customer. In the United States, many enterprises use the social-security number of a person (a unique number the U.S. government assigns to every person in the United States) as a customer identifier.

A **relationship** is an association among several entities. For example, a *depositor* relationship associates a customer with each account that she has. The set of all entities of

the same type and the set of all relationships of the same type are termed an **entity set** and **relationship set**, respectively.

The overall logical structure (schema) of a database can be expressed graphically by an *E-R diagram.*

1.8.2 Relational Model

The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name.

The data is arranged in a relation which is visually represented in a two dimensional table. The data is inserted into the table in the form of tuples (which are nothing but rows). A tuple is formed by one or more than one attributes, which are used as basic building blocks in the formation of various expressions that are used to derive a meaningful information. There can be any number of tuples in the table, but all the tuple contain fixed and same attributes with varying values. The relational model is implemented in database where a relation is represented by a table, a tuple is represented by a row, an attribute is represented by a column of the table, attribute name is the name of the column such as ‘identifier’, ‘name’, ‘city’ etc., attribute value contains the value for column in the row. Constraints are applied to the table and form the logical schema. In order to facilitate the selection of a particular row/tuple from the table, the attributes i.e. column names are used, and to expedite the selection of the rows some fields are defined uniquely to use them as indexes, this helps in searching the required data as fast as possible. All the relational algebra operations, such as Select, Intersection, Product, Union, Difference, Project, Join, Division, Merge etc. can also be performed on the Relational Database Model. Operations on the Relational Database Model are facilitated with the help of different conditional expressions, various key attributes, pre-defined constraints etc.

1.8.3 Other Data Models

The **object-oriented data model** is another data model that has seen increasing attention. The object-oriented model can be seen as extending the E-R model with notions object-oriented data model.

The **object-relational data model** combines features of the object-oriented data

model and relational data model. Semistructured data models permit the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast with the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The **extensible markup language (XML)** is widely used to represent semistructured data.

Historically, two other data models, the **network data model** and the **hierarchical data model**, preceded the relational data model. These models were tied closely to the underlying implementation, and complicated the task of modeling data. As a result they are little used now, except in old database code that is still in service in some places. They are outlined in Appendices A and B, for interested readers.

1.9 Database Languages

A database system provides a **data definition language** to specify the database schema and a **data manipulation language** to express database queries and updates. In practice, the data definition and data manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language.

1.9.1 Data-Definition Language

We specify a database schema by a set of definitions expressed by a special language called a **data-definition language** (**DDL**).

For instance, the following statement in the SQL language defines the *account* table:

create table *account (account-number* ***char****(10), balance* ***integer****)*

Execution of the above DDL statement creates the *account* table. In addition, it updates a special set of tables called the **data dictionary** or **data directory**.

A data dictionary contains **metadata**—that is, data about data. The schema of a table is an example of metadata. A database system consults the data dictionary before reading or modifying actual data.

We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a **data storage and definition** language. These statements define the implementation details of the database schemas, which are usually hidden from the users.

The data values stored in the database must satisfy certain **consistency constraints**. For example, suppose the balance on an account should not fall below $100. The DDL provides facilities to specify such constraints. The database systems check these constraints every time the database is updated.

1.9.2 Data-Manipulation Language

Data manipulation is

The retrieval of information stored in the database

The insertion of new information into the database

The deletion of information from the database

The modification of information stored in the database

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organized by the appropriate data model. There are basically two types:

Procedural DMLs require a user to specify *what* data are needed and *how* to get those data.

Declarative DMLs (also referred to as **nonprocedural** DMLs) require a user to specify *what* data are needed *without* specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data. The DML component of the SQL language is nonprocedural.

A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. Although technically incorrect, it is common practice to use the terms *query language* and *data manipulation language* synonymously.

This query in the SQL language finds the name of the customer whose customer-id is 192-83-7465:

select *customer.customer-name*
from *customer*
where *customer.customer-id* = 192-83-7465

The query specifies that those rows *from* the table *customer where* the *customer-id* is 192-83-7465 must be retrieved, and the *customer-name* attribute of these rows must be displayed.

Queries may involve information from more than one table. For instance, the following query finds the balance of all accounts owned by the customer with customerid 192-83-7465.

select *account.balance*
from *depositor, account*
where *depositor.customer-id* = 192-83-7465 **and**
depositor.account-number = account.account-number

There are a number of database query languages in use, either commercially or experimentally.

The levels of abstraction apply not only to defining or structuring data, but also to manipulating data. At the physical level, we must define algorithms that allow efficient access to data. At higher levels of abstraction, we emphasize ease of use. The goal is to allow humans to interact efficiently with the system. The query processor component of the database system translates DML queries into sequences of actions at the physical level of the database system.

1.10 Data Dictionary

We can define a data dictionary as a DBMS component that stores the definition of data characteristics and relationships. You may recall that such "data about data" were labeled metadata. The DBMS data dictionary provides the DBMS with its self describing characteristic. In effect, the data dictionary resembles and X-ray of the company's entire data set, and is a crucial element in the data administration function.

The two main types of data dictionary exist, integrated and stand alone. An integrated data dictionary is included with the DBMS. For example, all relational DBMSs include a built in data dictionary or system catalog that is frequently accessed and updated by the RDBMS. Other DBMSs especially older types, do not have a built in data dictionary instead the DBA may use third party stand alone data dictionary systems.

Data dictionaries can also be classified as active or passive. An active data dictionary is automatically updated by the DBMS with every database access, thereby keeping its

access information up-to-date. A passive data dictionary is not updated automatically and usually requires a batch process to be run. Data dictionary access information is normally used by the DBMS for query optimization purpose.

The data dictionary's main function is to store the description of all objects that interact with the database. Integrated data dictionaries tend to limit their metadata to the data managed by the DBMS. Stand alone data dictionary systems are more usually more flexible and allow the DBA to describe and manage all the organization's data, whether or not they are computerized. Whatever the data dictionary's format, its existence provides database designers and end users with a much improved ability to communicate. In addition, the data dictionary is the tool that helps the DBA to resolve data conflicts.

Although, there is no standard format for the information stored in the data dictionary several features are common. For example, the data dictionary typically stores descriptions of all:

- Data elements that are define in all tables of all databases. Specifically the data dictionary stores the name, datatypes, display formats, internal storage formats, and validation rules. The data dictionary tells where an element is used, by whom it is used and so on.
- Tables define in all databases. For example, the data dictionary is likely to store the name of the table creator, the date of creation access authorizations, the number of columns, and so on.
- Indexes define for each database tables. For each index the DBMS stores at least the index name the attributes used, the location, specific index characteristics and the creation date.
- Define databases: who created each database, the date of creation where the database is located, who the DBA is and so on.
- End users and The Administrators of the data base
- Programs that access the database including screen formats, report formats application formats, SQL queries and so on.
- Access authorization for all users of all databases.
- Relationships among data elements which elements are involved: whether the relationship are mandatory or optional, the connectivity and cardinality and so on.

If the data dictionary can be organized to include data external to the DBMS itself, it becomes an especially flexible to for more general corporate resource management. The management of such an extensive data dictionary, thus, makes it possible to manage the use and allocation of all of the organization information regardless whether it has its roots in the database data. This is why some managers consider the data dictionary to be the key element of the information resource management function. And this is also why the data dictionary might be described as the information resource dictionary.

The metadata stored in the data dictionary is often the bases for monitoring the database use and assignment of access rights to the database users. The information stored in the database is usually based on the relational table format, thus , enabling the DBA to query the database with SQL command. For example, SQL command can be used to extract information about the users of the specific table or about the access rights of a particular users.

1.11 Database Administrators and Database Users

A primary goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users or database administrators.

1.11.1 Database Users and User Interfaces

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.

Naive users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller who needs to transfer $50 from account *A* to account *B* invokes a program called *transfer*. This program asks the teller for the amount of money to be transferred, the account from which the money is to be transferred, and the account to which the money is to be transferred.

As another example, consider a user who wishes to find her account balance over the World Wide Web. Such a user may access a form, where she enters her account number. An application program at the Web server then retrieves the account balance, using the given account number, and passes this information back to the user. The typical user

interface for naive users is a forms interface, where the user can fill in appropriate fields of the form. Naive users may also simply read *reports* generated from the database.

Application programmers are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reports without writing a program. There are also special types of programming languages that combine imperative control structures (for example, for loops, while loops and if-then-else statements) with statements of the data manipulation language. These languages, sometimes called *fourth-generation languages*, often include special features to facilitate the generation of forms and the display of data on the screen. Most major commercial database systems include a fourth generation language.

Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a **query processor**, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category.

Online analytical processing (OLAP) tools simplify analysts' tasks by letting them view summaries of data in different ways. For instance, an analyst can see total sales by region (for example, North, South, East, and West), or by product, or by a combination of region and product (that is, total sales of each product in each region). The tools also permit the analyst to select specific regions, look at data in more detail (for example, sales by city within a region) or look at the data in less detail (for example, aggregate products together by category).

Another class of tools for analysts is **data mining** tools, which help them find certain kinds of patterns in data.

Specialized users are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework.

Among these applications are computer-aided design systems, knowledge base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems.

1.11.2 Database Administrator

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a **database administrator** (**DBA**). The functions of a DBA include:

Schema definition. The DBA creates the original database schema by executing a set of data definition statements in the DDL.

Storage structure and access-method definition.

Schema and physical-organization modification. The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.

Granting of authorization for data access. By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.

Routine maintenance. Examples of the database administrator's routine maintenance activities are:

Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.

Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.

Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

1.12 DBMS Architecture and Data Independence

Three important characteristics of the database approach are (1) insulation of programs and data (program-data and program-operation independence); (2) support of multiple user views; and (3) use of a catalog to store the database description (schema). In this section we specify an architecture for database systems, called the **three-schema architecture**, which was proposed to help achieve and visualize these characteristics. We then discuss the concept of data independence.

1.12.1 The Three-Schema Architecture

The goal of the three-schema architecture, illustrated in Figure 1.1, is to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels:

1. The **internal level** has an **internal schema,** which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The **conceptual level** has a **conceptual schema,** which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.
3. The **external** or **view level** includes a number of **external schemas** or **user views.** Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

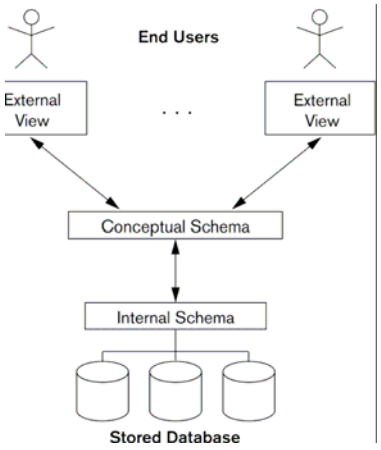


Figure 1.1 The Three Schema Architecture

The three-schema architecture is a convenient tool for the user to visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely, but support the three-schema architecture to some extent. Some DBMSs may include physical-level details in the conceptual schema. In most DBMSs that support user views,

external schemas are specified in the same data model that describes the conceptual-level information. Some DBMSs allow different data models to be used at the conceptual and external levels.

Notice that the three schemas are only *descriptions* of data; the only data that *actually* exists is at the physical level. In a DBMS based on the three-schema architecture, each user group refers only to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called **mappings.** These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

1.12.2 Data Independence

The three-schema architecture can be used to explain the concept of **data independence,** which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), or to reduce the database (by removing a record type or data item). In the latter case, external schemas that refer only to the remaining data should not be affected. Only the view definition and the mappings need be changed in a DBMS that supports logical data independence. Application programs that reference the external schema constructs must work as before, after the conceptual schema undergoes a logical reorganization. Changes to constraints can be applied also to the conceptual schema without affecting the external schemas or application programs.

2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual (or external) schemas. Changes to the internal schema may be needed because some physical files had to be reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema.

Whenever we have a multiple-level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. The DBMS uses additional software to accomplish these mappings by referring to the mapping information in the catalog. Data independence is accomplished because, when the schema is changed at some level, the schema at the next higher level remains unchanged; only the *mapping* between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed.

The three-schema architecture can make it easier to achieve true data independence, both physical and logical. However, the two levels of mappings create an overhead during compilation or execution of a query or program, leading to inefficiencies in the DBMS. Because of this, few DBMSs have implemented the full three-schema architecture.

1.13 Types of Database System

Several criteria are normally used to classify DBMSs. The *first* is the data model on which the DBMS is based. The main data model used in many current commercial DBMSs is the relational data model. The object data model was implemented in some commercial systems but has not had widespread use. Many legacy (older) applications still run on database systems based on the hierarchical and network data models. The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed in object databases. This has led to a new class of DBMSs called object-relational DBMSs. We can hence categorize DBMSs based on the *data model*: **relational, object, object-relational, hierarchical, network, and other**.

The *second* criterion used to classify DBMSs is the number of users supported by the system. **Single-user systems** support only one user at a time and are mostly used with personal computers. **Multiuser systems**, which include the majority of DBMSs, support multiple users concurrently. A *third* criterion is the number of sites over which the

database is distributed. A DBMS is centralized if the data is stored at a single computer site. A **centralized DBMS** can support multiple users, but the DBMS and the database themselves reside totally at a single computer site. A **distributed DBMS** (DDBMS) can have the actual database and DBMS software distributed over many sites, connected by a computer network. Homogeneous DDBMSs use the same DBMS software at multiple sites. A recent trend is to develop software to access several autonomous preexisting databases stored under heterogeneous llBMSs. This leads to a federated DBMS (or multidatabase system), in which the participating DBMSs are loosely coupled and have a degree of local autonomy. Many DBMSs use a client-server architecture.

1.14 Summary

In this chapter we have discussed in a relatively informal manner the major components of a database system. We summarise the discussion below:

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data. This is a collection of related data with an implicit meaning and hence is a database.

A **datum** – a unit of data – is a symbol or a set of symbols which is used to represent something. This relationship between symbols and what they represent is the essence of what we mean by **information**.

Knowledge refers to the practical use of information.

The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**.

The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level.Adatabase may also have several schemas at the view level, sometimes called **subschemas**, that describe different views of the database.

Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

A database system provides a **data definition language** to specify the database schema and a **data manipulation language** to express database queries and updates.

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a **database administrator** (**DBA**).

1.15 Key Words

DBMS, Data Integrity, Data Persistence, Instances, Schemas, Physical Schema, Logical Schema, Data Model, DDL, DML, Data Dictionary

1.16 Self Assessment Questions

1. Why would you choose a database system instead of simply storing data in operating system files? When would it make sense *not* to use a database system?
2. What is logical data independence and why is it important?
3. Explain the difference between logical and physical data independence.
4. Explain the difference between external, internal, and conceptual schemas. How are these different schema layers related to the concepts of logical and physical data independence?
5. What are the responsibilities of a DBA?
6. Distinguish between logical and physical database design.
7. Describe and define the key properties of a database system. Give an organizational example of the benefits of each property.

1.17 References/Suggested Readings

1 http://www.microsoft-accesssolutions.co.uk

2 Date, C, J, Introduction to Database Systems, 7th edition

3 Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld.

MCA 202/MS 11

Author: Abhishek Taneja **Vetter: Sh. Dharminder Kumar**
Lesson No. : 02 **Lesson: Data Modeling Using Entity-Relationship Approach**

Structure

2.0 Objectives

At the end of this chapter the reader will be able to:

- Describe basic concepts of ER Model
- Describe components of a data model
- Describe basic constructs of E-R Modeling
- Describe data modeling as a part of database design process
- Describe steps in building the data model
- Describe developing the basic schema

2.1 Introduction

A data model is a conceptual representation of the data structures that are required by a database. The data structures include the data objects, the associations between data objects, and the rules which govern operations on the objects. As the name implies, the data model focuses on what data is required and how it should be organized rather than what operations will be performed on the data. To use a common analogy, the data model is equivalent to an architect's building plans.

A data model is independent of hardware or software constraints. Rather than try to represent the data as a database would see it, the data model focuses on representing the data as the user sees it in the "real world". It serves as a bridge between the concepts that make up real-world events and processes and the physical representation of those concepts in a database.

Methodology

There are two major methodologies used to create a data model: the Entity-Relationship (ER) approach and the Object Model. This document uses the Entity-Relationship approach.

2.2 Data Modeling In the Context of Database Design

Database design is defined as: "design the logical and physical structure of one or more databases to accommodate the information needs of the users in an organization for a defined set of applications". The design process roughly follows five steps:

1. Planning and analysis
2. Conceptual design
3. Logical design
4. Physical design
5. Implementation

The data model is one part of the conceptual design process. The other, typically is the functional model. The data model focuses on what data should be stored in the database while the functional model deals with how the data is processed. To put this in the context of the relational database, the data model is used to design the relational tables. The functional model is used to design the queries which will access and perform operations on those tables.

Components of a Data Model

The data model gets its inputs from the planning and analysis stage. Here the modeler, along with analysts, collects information about the requirements of the database by reviewing existing documentation and interviewing end-users.

The data model has two outputs. The first is an entity-relationship diagram which represents the data structures in a pictorial form. Because the diagram is easily learned, it is valuable tool to communicate the model to the end-user. The second component is a data document. This a document that describes in detail the data objects, relationships, and rules required by the database. The dictionary provides the detail required by the database developer to construct the physical database.

Why is Data Modeling Important?

Data modeling is probably the most labor intensive and time consuming part of the development process. Why bother especially if you are pressed for time? A common response by practitioners who write on the subject is that you should no more build a database without a model than you should build a house without blueprints.

The goal of the data model is to make sure that the all data objects required by the database are completely and accurately represented. Because the data model uses easily understood notations and natural language , it can be reviewed and verified as correct by the end-users.

The data model is also detailed enough to be used by the database developers to use as a "blueprint" for building the physical database. The information contained in the data model will be used to define the relational tables, primary and foreign keys, stored procedures, and triggers. A poorly designed database will require more time in the long-term. Without careful planning you may create a database that omits data required to create critical reports, produces results that are incorrect or inconsistent, and is unable to accommodate changes in the user's requirements.

2.3 The Entity-Relationship Model

The Entity-Relationship (ER) model was originally proposed by Peter in 1976 as a way to unify the network and relational database views. Simply stated the ER model is a conceptual data model that views the real world as entities and relationships. A basic component of the model is the Entity-Relationship diagram which is used to visually

represents data objects. Since Chen wrote his paper the model has been extended and today it is commonly used for database design For the database designer, the utility of the ER model is:

It maps well to the relational model. The constructs used in the ER model can easily be transformed into relational tables.

It is simple and easy to understand with a minimum of training. Therefore, the model can be used by the database designer to communicate the design to the end user.

In addition, the model can be used as a design plan by the database developer to implement a data model in a specific database management software.

Basic Constructs of E-R Modeling

The ER model views the real world as a construct of entities and association between entities.

Entities

Entities are the principal data object about which information is to be collected. Entities are usually recognizable concepts, either concrete or abstract, such as person, places, things, or events which have relevance to the database. Some specific examples of entities are EMPLOYEES, PROJECTS, INVOICES. An entity is analogous to a table in the relational model.

Entities are classified as independent or dependent (in some methodologies, the terms used are strong and weak, respectively). An independent entity is one that does not rely on another for identification. A dependent entity is one that relies on another for identification.

An entity occurrence (also called an instance) is an individual occurrence of an entity. An occurrence is analogous to a row in the relational table.

Special Entity Types

Associative entities (also known as intersection entities) are entities used to associate two or more entities in order to reconcile a many-to-many relationship.

Subtypes entities are used in generalization hierarchies to represent a subset of instances of their parent entity, called the supertype, but which have attributes or relationships that apply only to the subset.

Associative entities and generalization hierarchies are discussed in more detail below.

Relationships

A Relationship represents an association between two or more entities. An example of a relationship would be:

Employees are assigned to projects

Projects have subtasks

Departments manage one or more projects

Relationships are classified in terms of degree, connectivity, cardinality, and existence. These concepts will be discussed below.

Attributes

Attributes describe the entity of which they are associated. A particular instance of an attribute is a value. For example, "Jane R. Hathaway" is one value of the attribute Name. The domain of an attribute is the collection of all possible values an attribute can have. The domain of Name is a character string.

Attributes can be classified as identifiers or descriptors. Identifiers, more commonly called keys, uniquely identify an instance of an entity. A descriptor describes a non-unique characteristic of an entity instance.

Classifying Relationships

Relationships are classified by their degree, connectivity, cardinality, direction, type, and existence. Not all modeling methodologies use all these classifications.

Degree of a Relationship

The degree of a relationship is the number of entities associated with the relationship. The n-ary relationship is the general form for degree n. Special cases are the binary, and ternary ,where the degree is 2, and 3, respectively.

Binary relationships, the association between two entities is the most common type in the real world. A recursive binary relationship occurs when an entity is related to itself. An example might be "some employees are married to other employees".

A ternary relationship involves three entities and is used when a binary relationship is inadequate. Many modeling approaches recognize only binary relationships. Ternary or n-ary relationships are decomposed into two or more binary relationships.

Connectivity and Cardinality The connectivity of a relationship describes the mapping of associated entity instances in the relationship. The values of connectivity are "one" or

"many". The cardinality of a relationship is the actual number of related occurences for each of the two entities. The basic types of connectivity for relations are: one-to-one, one-to-many, and many-to-many.

A *one-to-one* (1:1) relationship is when at most one instance of a entity A is associated with one instance of entity B. For example, "employees in the company are each assigned their own office. For each employee there exists a unique office and for each office there exists a unique employee.

A *one-to-many* (1:N) relationships is when for one instance of entity A, there are zero, one, or many instances of entity B, but for one instance of entity B, there is only one instance of entity A. An example of a 1:N relationships is

A department has many employees

Each employee is assigned to one department

A *many-to-many* (M:N) relationship, sometimes called non-specific, is when for one instance of entity A, there are zero, one, or many instances of entity B and for one instance of entity B there are zero, one, or many instances of entity A. An example is:

employees can be assigned to no more than two projects at the same time;

projects must have assigned at least three employees

A single employee can be assigned to many projects; conversely, a single project can have assigned to it many employee. Here the cardinality for the relationship between employees and projects is two and the cardinality between project and employee is three. Many-to-many relationships cannot be directly translated to relational tables but instead must be transformed into two or more one-to-many relationships using associative entities.

Direction

The direction of a relationship indicates the originating entity of a binary relationship. The entity from which a relationship originates is the parent entity; the entity where the relationship terminates is the child entity.

The direction of a relationship is determined by its connectivity. In a one-to-one relationship the direction is from the independent entity to a dependent entity. If both entities are independent, the direction is arbitrary. With one-to-many relationships, the

entity occurring once is the parent. The direction of many-to-many relationships is arbitrary.

Type

An *identifying relationship* is one in which one of the child entities is also a dependent entity. A *non-identifying relationship* is one in which both entities are independent.

Existence

Existence denotes whether the existence of an entity instance is dependent upon the existence of another, related, entity instance. The existence of an entity in a relationship is defined as either *mandatory* or *optional*. If an instance of an entity must always occur for an entity to be included in a relationship, then it is mandatory. An example of mandatory existence is the statement "every project must be managed by a single department". If the instance of the entity is not required, it is optional. An example of optional existence is the statement, "employees may be assigned to work on projects".

Generalization Hierarchies

A generalization hierarchy is a form of abstraction that specifies that two or more entities that share common attributes can be generalized into a higher level entity type called a *supertype* or *generic entity*. The lower-level of entities become the subtype, or categories, to the supertype. Subtypes are dependent entities.

Generalization occurs when two or more entities represent categories of the same real-world object. For example, Wages_Employees and Classified_Employees represent categories of the same entity, Employees. In this example, Employees would be the supertype; Wages_Employees and Classified_Employees would be the subtypes.

Subtypes can be either mutually exclusive (disjoint) or overlapping (inclusive). A mutually exclusive category is when an entity instance can be in only one category. The above example is a mutually exclusive category. An employee can either be wages or classified but not both. An overlapping category is when an entity instance may be in two or more subtypes. An example would be a person who works for a university could also be a student at that same university. The completeness constraint requires that all instances of the subtype be represented in the supertype. Generalization hierarchies can be nested. That is, a subtype of one hierarchy can be a supertype of another. The level of

nesting is limited only by the constraint of simplicity. Subtype entities may be the parent entity in a relationship but not the child.

ER Notation

There is no standard for representing data objects in ER diagrams. Each modeling methodology uses its own notation. All notational styles represent entities as rectangular boxes and relationships as lines connecting boxes. Each style uses a special set of symbols to represent the cardinality of a connection. The notation used in this document is from Martin. The symbols used for the basic ER constructs are:

- Entities are represented by labeled rectangles. The label is the name of the entity. Entity names should be singular nouns.
- Relationships are represented by a solid line connecting two entities. The name of the relationship is written above the line. Relationship names should be verbs.
- Attributes, when included, are listed inside the entity rectangle. Attributes which are identifiers are underlined. Attribute names should be singular nouns.
- Cardinality of many is represented by a line ending in a crow's foot. If the crow's foot is omitted, the cardinality is one.
- Existence is represented by placing a circle or a perpendicular bar on the line. Mandatory existence is shown by the bar (looks like a 1) next to the entity for an instance is required. Optional existence is shown by placing a circle next to the entity that is optional.

Examples of these symbols are shown in Figure 2.1 below:

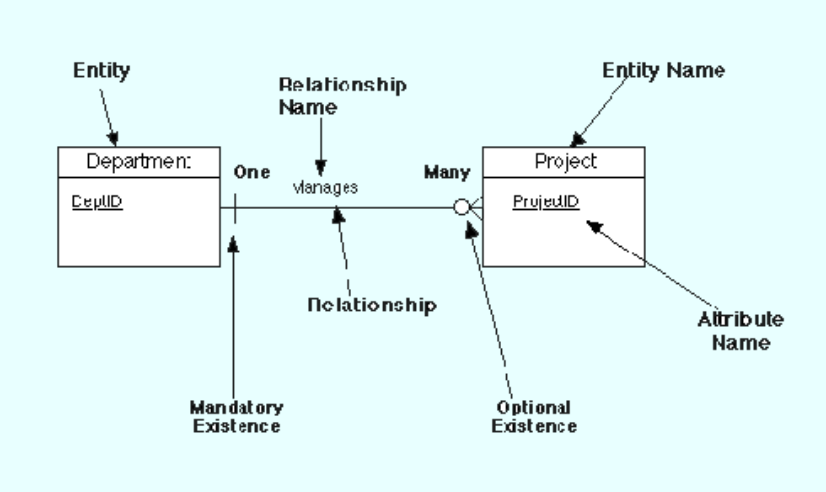


Figure 2.1 ER Notation

2.4 Data Modeling As Part of Database Design

The data model is one part of the conceptual design process. The other is the **function model.** The data model focuses on what data should be stored in the database while the function model deals with how the data is processed. To put this in the context of the relational database, the data model is used to design the relational tables. The functional model is used to design the queries that will access and perform operations on those tables.

Data modeling is preceeded by planning and analysis. The effort devoted to this stage is proportional to the scope of the database. The planning and analysis of a database intended to serve the needs of an enterprise will require more effort than one intended to serve a small workgroup.

The information needed to build a data model is gathered during the requirments analysis. Although not formally considered part of the data modeling stage by some methodologies, in reality the requirements analysis and the ER diagramming part of the data model are done at the same time.

Requirements Analysis

The goals of the requirements analysis are:

- To determine the data requirements of the database in terms of primitive objects
- To classify and describe the information about these objects
- To identify and classify the relationships among the objects
- To determine the types of transactions that will be executed on the database and the interactions between the data and the transactions
- To identify rules governing the integrity of the data

The modeler, or modelers, works with the end users of an organization to determine the data requirements of the database. Information needed for the requirements analysis can be gathered in several ways:

Review of existing documents - such documents include existing forms and reports, written guidelines, job descriptions, personal narratives, and memoranda. Paper documentation is a good way to become familiar with the organization or activity you need to model.

Interviews with end users - these can be a combination of individual or group meetings. Try to keep group sessions to under five or six people. If possible, try to have everyone with the same function in one meeting. Use a blackboard, flip charts, or overhead transparencies to record information gathered from the interviews.

Review of existing automated systems - if the organization already has an automated system, review the system design specifications and documentation

The requirements analysis is usually done at the same time as the data modeling. As information is collected, data objects are identified and classified as either entities, attributes, or relationship; assigned names; and, defined using terms familiar to the end-users. The objects are then modeled and analysed using an ER diagram. The diagram can be reviewed by the modeler and the end-users to determine its completeness and accuracy. If the model is not correct, it is modified, which sometimes requires additional information to be collected. The review and edit cycle continues until the model is certified as correct.

Three points to keep in mind during the requirements analysis are:

1. Talk to the end users about their data in "real-world" terms. Users do not think in terms of entities, attributes, and relationships but about the actual people, things, and activities they deal with daily.
2. Take the time to learn the basics about the organization and its activities that you want to model. Having an understanding about the processes will make it easier to build the model.
3. End-users typically think about and view data in different ways according to their function within an organization. Therefore, it is important to interview the largest number of people that time permits.

2.5 Steps In Building the Data Model

While ER model lists and defines the constructs required to build a data model, there is no standard process for doing so. Some methodologies, such as IDEFIX, specify a bottom-up development process were the model is built in stages. Typically, the entities and relationships are modeled first, followed by key attributes, and then the model is finished by adding non-key attributes. Other experts argue that in practice, using a phased

approach is impractical because it requires too many meetings with the end-users. The sequence used for this document are:

1. Identification of data objects and relationships
2. Drafting the initial ER diagram with entities and relationships
3. Refining the ER diagram
4. Add key attributes to the diagram
5. Adding non-key attributes
6. Diagramming Generalization Hierarchies
7. Validating the model through normalization
8. Adding business and integrity rules to the Model

In practice, model building is not a strict linear process. As noted above, the requirements analysis and the draft of the initial ER diagram often occur simultaneously. Refining and validating the diagram may uncover problems or missing information which require more information gathering and analysis

Identifying Data Objects and Relationships

In order to begin constructing the basic model, the modeler must analyze the information gathered during the requirements analysis for the purpose of:

- Classifying data objects as either entities or attributes
- Identifying and defining relationships between entities
- Naming and defining identified entities, attributes, and relationships
- Documenting this information in the data document

To accomplish these goals the modeler must analyze narratives from users, notes from meeting, policy and procedure documents, and, if lucky, design documents from the current information system.

Although it is easy to define the basic constructs of the ER model, it is not an easy task to distinguish their roles in building the data model. What makes an object an entity or attribute? For example, given the statement "employees work on projects". Should employees be classified as an entity or attribute? Very often, the correct answer depends upon the requirements of the database. In some cases, employee would be an entity, in some it would be an attribute.

While the definitions of the constructs in the ER Model are simple, the model does not address the fundamental issue of how to identify them. Some commonly given guidelines are:

- Entities contain descriptive information
- Attributes either identify or describe entities
- Relationships are associations between entities

These guidelines are discussed in more detail below.

- Entities
- Attributes
    - Validating Attributes
    - Derived Attributes and Code Values
- Relationships
- Naming Data Objects
- Object Definition
- Recording Information in Design Document

Entities

There are various definitions of an entity:

"Any distinguishable person, place, thing, event, or concept, about which information is kept"

"A thing which can be distinctly identified"

"Any distinguishable object that is to be represented in a database"

"...anything about which we store information (e.g. supplier, machine tool, employee, utility pole, airline seat, etc.). For each entity type, certain attributes are stored".

These definitions contain common themes about entities:

- An entity is a "thing", "concept" or, object". However, entities can sometimes represent the relationships between two or more objects. This type of entity is known as an associative entity.
- Entities are objects which contain descriptive information. If an data object you have identified is described by other objects, then it is an entity. If there is no descriptive information associated with the item, it is not an entity. Whether or

not a data object is an entity may depend upon the organization or activity being modeled.

- An entity represents many things which share properties. They are not single things. For example, King Lear and Hamlet are both plays which share common attributes such as name, author, and cast of characters. The entity describing these things would be PLAY, with King Lear and Hamlet being instances of the entity.
- Entities which share common properties are candidates for being converted to generalization hierarchies (See below)
- Entities should not be used to distinguish between time periods. For example, the entities 1st Quarter Profits, 2nd Quarter Profits, etc. should be collapsed into a single entity called Profits. An attribute specifying the time period would be used to categorize by time
- Not every thing the users want to collect information about will be an entity. A complex concept may require more than one entity to represent it. Others "things" users think important may not be entities.

Attributes

Attributes are data objects that either identify or describe entities. Attributes that identify entities are called *key attributes*. Attributes that describe an entity are called non-key attributes. Key attributes will be discussed in detail in a latter section.

The process for identifying attributes is similar except now you want to look for and extract those names that appear to be descriptive noun phrases.

Validating Attributes

Attribute values should be *atomic*, that is, present a single fact. Having disaggregated data allows simpler programming, greater reusability of data, and easier implementation of changes. Normalization also depends upon the "single fact" rule being followed. Common types of violations include:

- Simple aggregation - a common example is Person Name which concatenates first name, middle initial, and last name. Another is Address which concatenates, street address, city, and zip code. When dealing with such attributes, you need to find out if there are good reasons for decomposing them. For example, do the end-

users want to use the person's first name in a form letter? Do they want to sort by zip code?

- Complex codes - these are attributes whose values are codes composed of concatenated pieces of information. An example is the code attached to automobiles and trucks. The code represents over 10 different pieces of information about the vehicle. Unless part of an industry standard, these codes have no meaning to the end user. They are very difficult to process and update.
- Text blocks - these are free-form text fields. While they have a legitimate use, an over reliance on them may indicate that some data requirements are not met by the model.
- Mixed domains - this is where a value of an attribute can have different meaning under different conditions

Derived Attributes and Code Values

Two areas where data modeling experts disagree is whether derived attributes and attributes whose values are codes should be permitted in the data model.

Derived attributes are those created by a formula or by a summary operation on other attributes. Arguments against including derived data are based on the premise that derived data should not be stored in a database and therefore should not be included in the data model. The arguments in favor are:

- Derived data is often important to both managers and users and therefore should be included in the data model
- It is just as important, perhaps more so, to document derived attributes just as you would other attributes
- Including derived attributes in the data model does not imply how they will be implemented

A coded value uses one or more letters or numbers to represent a fact. For example, the value Gender might use the letters "M" and "F" as values rather than "Male" and "Female". Those who are against this practice cite that codes have no intuitive meaning to the end-users and add complexity to processing data. Those in favor argue that many organizations have a long history of using coded attributes, that codes save space, and

improve flexibility in that values can be easily added or modified by means of look-up tables.

Relationships

Relationships are associations between entities. Typically, a relationship is indicated by a verb connecting two or more entities. For example:

employees are assigned to projects

As relationships are identified they should be classified in terms of cardinality, optionality, direction, and dependence. As a result of defining the relationships, some relationships may be dropped and new relationships added. Cardinality quantifies the relationships between entities by measuring how many instances of one entity are related to a single instance of another. To determine the cardinality, assume the existence of an instance of one of the entities. Then determine how many specific instances of the second entity could be related to the first. Repeat this analysis reversing the entities. For example:

Employees may be assigned to no more than three projects at a time; every project has at least two employees assigned to it.

Here the cardinality of the relationship from employees to projects is three; from projects to employees, the cardinality is two. Therefore, this relationship can be classified as a many-to-many relationship.

If a relationship can have a cardinality of zero, it is an optional relationship. If it must have a cardinality of at least one, the relationship is mandatory. Optional relationships are typically indicated by the conditional tense. For example:

An employee **may** be assigned to a project

Mandatory relationships, on the other hand, are indicated by words such as must have. For example:

A student **must** register for at least three course each semester

In the case of the specific relationship form (1:1 and 1:M), there is always a parent entity and a child entity. In one-to-many relationships, the parent is always the entity with the cardinality of one. In one-to-one relationships, the choice of the parent entity must be made in the context of the business being modeled. If a decision cannot be made, the choice is arbitrary.

Naming Data Objects

The names should have the following properties:

- Unique
- Have meaning to the end-user
- Contain the minimum number of words needed to uniquely and accurately describe the object

For entities and attributes, names are singular nouns while relationship names are typically verbs.

Some authors advise against using abbreviations or acronyms because they might lead to confusion about what they mean. Other believe using abbreviations or acronyms are acceptable provided that they are universally used and understood within the organization.

You should also take care to identify and resolve synonyms for entities and attributes. This can happen in large projects where different departments use different terms for the same thing.

Object Definition

Complete and accurate definitions are important to make sure that all parties involved in the modeling of the data know exactly what concepts the objects are representing.

Definitions should use terms familiar to the user and should precisely explain what the object represents and the role it plays in the enterprise. Some authors recommend having the end-users provide the definitions. If acronyms, or terms not universally understood are used in the definition, then these should be defined .

While defining objects, the modeler should be careful to resolve any instances where a single entity is actually representing two different concepts (homonyms) or where two different entities are actually representing the same "thing" (synonyms). This situation typically arises because individuals or organizations may think about an event or process in terms of their own function.

An example of a homonym would be a case where the Marketing Department defines the entity MARKET in terms of geographical regions while the Sales Departments thinks of this entity in terms of demographics. Unless resolved, the result would be an entity with two different meanings and properties.

Conversely, an example of a synonym would be the Service Department may have identified an entity called CUSTOMER while the Help Desk has identified the entity CONTACT. In reality, they may mean the same thing, a person who contacts or calls the organization for assistance with a problem. The resolution of synonyms is important in order to avoid redundancy and to avoid possible consistency or integrity problems.

Recording Information in Design Document

The design document records detailed information about each object used in the model. As you name, define, and describe objects, this information should be placed in this document. If you are not using an automated design tool, the document can be done on paper or with a word processor. There is no standard for the organization of this document but the document should include information about names, definitions, and, for attributes, domains.

Two documents used in the IDEF1X method of modeling are useful for keeping track of objects. These are the ENTITY-ENTITY matrix and the ENTITY-ATTRIBUTE matrix.

The ENTITY-ENTITY matrix is a two-dimensional array for indicating relationships between entities. The names of all identified entities are listed along both axes. As relationships are first identified, an "X" is placed in the intersecting points where any of the two axes meet to indicate a possible relationship between the entities involved. As the relationship is further classified, the "X" is replaced with the notation indicating cardinality.

The ENTITY-ATTRIBUTE matrix is used to indicate the assignment of attributes to entities. It is similar in form to the ENTITY-ENTITY matrix except attribute names are listed on the rows.

Figure 2.2 shows examples of an ENTITY-ENTITY matrix and an ENTITY-ATTRIBUTE matrix.

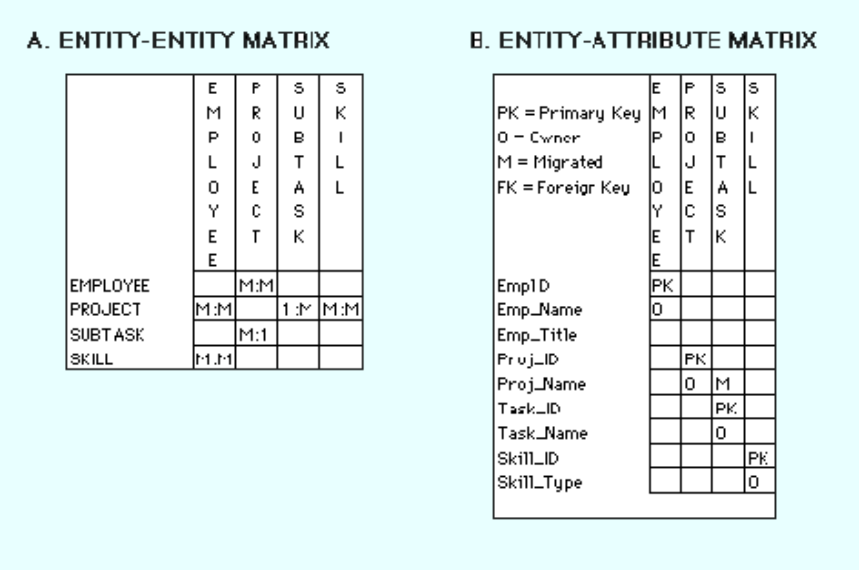


Figure 2.2

2.6 Developing the Basic Schema

Once entities and relationships have been identified and defined, the first draft of the entity relationship diagram can be created. This section introduces the ER diagram by demonstrating how to diagram binary relationships. Recursive relationships are also shown.

Binary Relationships

Figure 2.3 shows examples of how to diagram one-to-one, one-to-many, and many-to-many relationships.

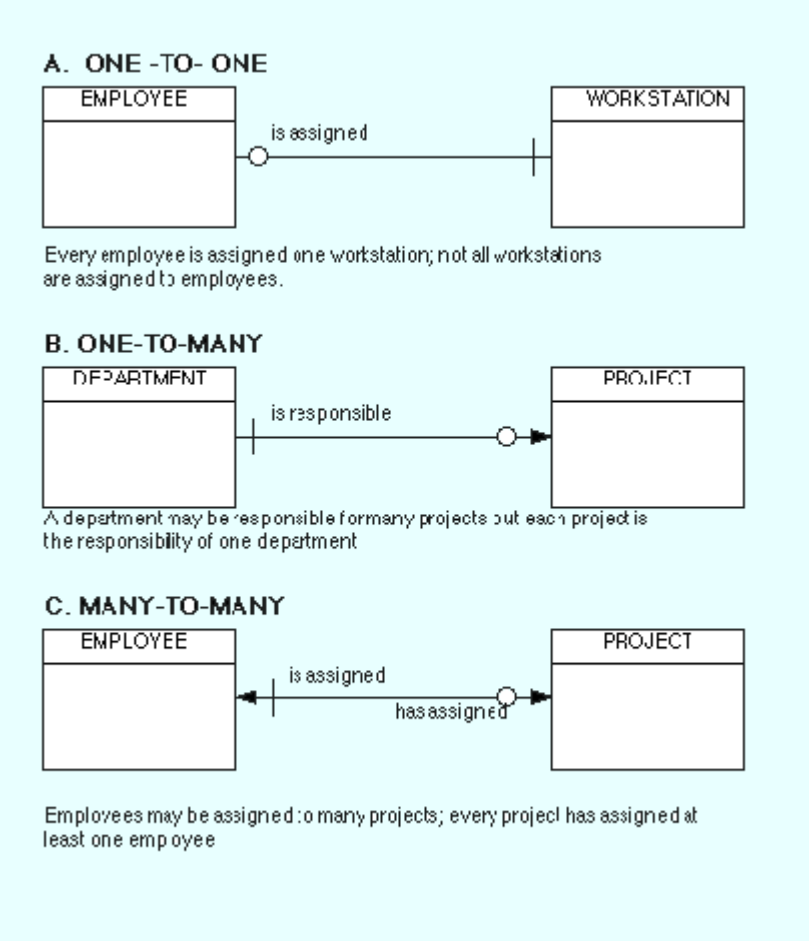


Figure 2.3 Example of Binary relationships

One-To-One

Figure 1A shows an example of a one-to-one diagram. Reading the diagram from left to right represents the relationship every employee is assigned a workstation. Because every employee must have a workstation, the symbol for mandatory existence—in this case the crossbar—is placed next to the WORKSTATION entity. Reading from right to left, the diagram shows that not all workstation are assigned to employees. This condition may reflect that some workstations are kept for spares or for loans. Therefore, we use the symbol for optional existence, the circle, next to EMPLOYEE. The cardinality and existence of a relationship must be derived from the "business rules" of the organization. For example, if all workstations owned by an organization were assigned to employees, then the circle would be replaced by a crossbar to indicate mandatory existence. One-to-one relationships are rarely seen in "real-world" data models. Some practioners advise

that most one-to-one relationships should be collapsed into a single entity or converted to a generalization hierarchy.

One-To-Many

Figure 1B shows an example of a one-to-many relationship between DEPARTMENT and PROJECT. In this diagram, DEPARTMENT is considered the parent entity while PROJECT is the child. Reading from left to right, the diagram represents departments may be responsible for many projects. The optionality of the relationship reflects the "business rule" that not all departments in the organization will be responsible for managing projects. Reading from right to left, the diagram tells us that every project must be the responsibility of exactly one department.

Many-To-Many

Figure 1C shows a many-to-many relationship between EMPLOYEE and PROJECT. An employee may be assigned to many projects; each project must have many employee Note that the association between EMPLOYEE and PROJECT is optional because, at a given time, an employee may not be assigned to a project. However, the relationship between PROJECT and EMPLOYEE is mandatory because a project must have at least two employees assigned. Many-To-Many relationships can be used in the initial drafting of the model but eventually must be transformed into two one-to-many relationships. The transformation is required because many-to-many relationships cannot be represented by the relational model. The process for resolving many-to-many relationships is discussed in the next section.

Recursive relationships

A recursive relationship is an entity is associated with itself. Figure 2.4 shows an example of the recursive relationship.

An employee may manage many employees and each employee is managed by one employee.

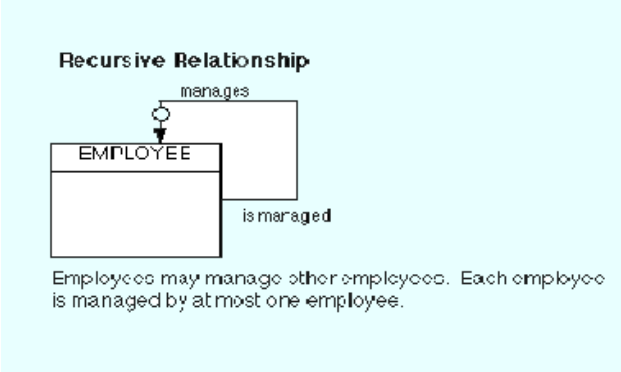


Figure 2.4 Example of Recursive relationship

2.7 Summary

- A data model is a plan for building a database. To be effective, it must be simple enough to communicate to the end user the data structure required by the database yet detailed enough for the database design to use to create the physical structure.
- The Entity-Relation Model (ER) is the most common method used to build data models for relational databases.
- The Entity-Relationship Model is a conceptual data model that views the real world as consisting of entities and relationships. The model visually represents these concepts by the Entity-Relationship diagram.
- The basic constructs of the ER model are entities, relationships, and attributes.
- Data modeling must be preceded by planning and analysis.
- Planning defines the goals of the database, explains why the goals are important, and sets out the path by which the goals will be reached.
- Analysis involves determining the requirements of the database. This is typically done by examining existing documentation and interviewing users.
- Data modeling is a bottom up process. A basic model, representing entities and relationships, is developed first. Then detail is added to the model by including information about attributes and business rules.

- The first step in creating the data model is to analyze the information gathered during the requirements analysis with the goal of identifying and classifying data objects and relationships
- The Entity-Relationship diagram provides a pictorial representation of the major data objects, the entities, and the relationships between them.

2.8 Key Words

ER Model, Database Design, Data Model, Schema, Entities, Relationship, Attributes, Cardinality

2.9 Self Assessment Questions

1. A university registrar's office maintains data about the following entities:
    - Courses, including number, title, credits, syllabus, and prerequisites;
    - Course offerings, including course number, year, semester, section number, instructor(s), timings, and classroom;
    - Students, including student-id, name, and program;
    - Instructors, including identification number, name, department, and title.

Further, the enrollment of students in courses and grades awarded to students in each course they are enrolled for must be appropriately modeled.Construct a E-R diagram for registrar's office. Document all assumptions that you make about the mapping constraints

2. Design an E-R diagram for keeping track of the exploits of your favorite sports team. You should store the matches played, the scores in each match, the players in each match, and individual player statistics for each match. Summary statistics should be modeled as derived attributes.
3. Explain the significance of ER Model for Database design?
4. Enumerate the basic constructs of ER Model

2.10 References/Suggested Readings

1. Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000
2. Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld
3. Elamasri R . and Navathe, S., Fundamentals of Database Systems (3rd Edition), Pearsson Education, 2000.

MCA 202/MS 11

Author: Abhishek Taneja **Vetter: Dr.Pradeep Bhatia**
Lesson: Relational Model **Lesson No. : 03**

Structure

3.0 Objectives

At the end of this chapter the reader will be able to:

- Describe Relational Model Concepts
- Describe properties of a relation and relational keys
- Describe relational model/integrity constraints
- Describe The Relational Algebra
- Introduce Oracle- A relational database management system

3.1 Introduction

The principles of the relational model were first outlined by Dr. E. F. Codd in a June 1970 paper called "A Relational Model of Data for Large Shared Data Banks:' In this paper. Dr. Codd proposed the relational model for database systems. The more popular models used at that time were hierarchical and network, or even simple flat file data structures. Relational database management systems (RDBMS) soon became very popular, especially for their ease of use and flexibility in structure. In addition, a number of innovative vendors, such as Oracle, supplemented the RDBMS \with a suite of powerful application development and user products, providing a total solution.

Earlier we saw how to convert an unorganized text description of information requirements into a conceptual design, by the use of ER diagrams. The advantage of ER diagrams is that they force you to identify data requirements that are implicitly known, but not explicitly written down in the original description. Here we will see how to convert this ER into a **logical design** (this will be defined below) of a relational database. The logical model is also called a **Relational Model**.

3.2 Relational Model Concepts

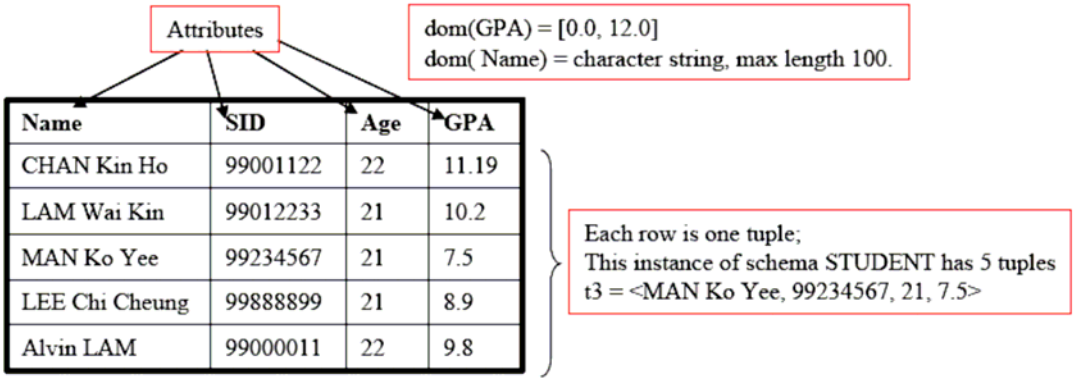
We shall represent a relation as a table with columns and rows. Each column of the **table** has a name, or ***attribute***. Each row is called a ***tuple***.

- **Domain**: a set of atomic values that an attribute can take
- **Attribute**: name of a column in a particular table (all data is stored in tables). Each attribute Ai must have a *domain*, dom(A_i).
- **Relational Schema**: The design of one table, containing the name of the table (i.e. the name of the relation), and the names of all the columns, or attributes.

Example: STUDENT(Name, SID, Age, GPA)

- **Degree of a Relation**: the number of attributes in the relation's schema.
- **Tuple, t, of R(A1, A2, A3, …, An):** an ORDERED set of values, < v1, v2, v3, …, vn>, where each v_i is a value from dom(A_i).

- **Relation Instance, r(R):** a set of tuples; thus, r(R) = { t1, t2, t3, …, tm}



Name	SID	Age	GPA
CHAN Kin Ho	99001122	22	11.19
LAM Wai Kin	99012233	21	10.2
MAN Ko Yee	99234567	21	7.5
LEE Chi Cheung	99888899	21	8.9
Alvin LAM	99000011	22	9.8

NOTES:

1. The tuples in an instance of a relation are not considered to be ordered putting the rows in a different sequence **does not change** the table.
2. Once the schema, R(A1, A2, A3, …, An) is defined, the values, vi, in each tuple, t, must be ordered as t = <v1, v2, v3, …, vn>

3.2.1 Properties of relations

Properties of database relations are:

- relation name is distinct from all other relations
- each cell of relation contains exactly one atomic (single) value
- each attribute has a distinct name
- values of an attribute are all from the same domain
- order of attributes has no significance
- each tuple is distinct; there are no duplicate tuples
- order of tuples has no significance, theoretically.

3.2.2 Relational keys

There are two kinds of keys in relations. The first are identifying keys: the **primary key** is the main concept, while two other keys – **super key** and **candidate key** – are related concepts. The second kind is the foreign key.

Identity Keys

Super Keys

A super key is a set of attributes whose values can be used to uniquely identify a tuple within a relation. A relation may have more than one super key, but it always has at least one: the set of all attributes that make up the relation.

Candidate Keys

A candidate key is a super key that is minimal; that is, there is no proper subset that is itself a superkey. A relation may have more than one candidate key, and the different candidate keys may have a different number of attributes. In other words, you should not interpret 'minimal' to mean the super key with the fewest attributes.

A candidate key has two properties:

(i) in each tuple of R, the values of K uniquely identify that tuple (uniqueness)

(ii) no proper subset of K has the uniqueness property (irreducibility).

Primary Key

The primary key of a relation is a candidate key especially selected to be the key for the relation. In other words, it is a choice, and there can be only one candidate key designated to be the primary key.

Relationship between identity keys

The relationship between keys:

Superkey $\supseteq$ Candidate Key $\supseteq$ Primary Key

Foreign keys

The attribute(s) within one relation that matches a candidate key of another relation. A relation may have several foreign keys, associated with different target relations.

Foreign keys allow users to link information in one relation to information in another relation. Without FKs, a database would be a collection of unrelated tables.

3.3 Relational Model Constraints

Integrity Constraints

Each relational schema must satisfy the following four types of constraints.

A. Domain constraints

Each attribute **Ai** must be an atomic value from dom(**Ai**) for that attribute.

The attribute, Name in the example is a BAD DESIGN (because sometimes we may want to search a person by only using their last name.

B. Key Constraints

Superkey of R: A set of attributes, SK, of R such that no two tuples in any valid relational instance, r(R), will have the same value for SK. Therefore, for any two distinct tuples, t1 and t2 in r(R),

t1[SK] != t2[SK].

Key of R: A minimal superkey. That is, a superkey, K, of R such that the removal of ANY attribute from K will result in a set of attributes that are not a superkey.

Example CAR(State, LicensePlateNo, VehicleID, Model, Year, Manufacturer)

This schema has two keys:

K1 = { State, LicensePlateNo}

K2 = { VehicleID }

Both K1 and K2 are superkeys.

K3 = { VehicleID, Manufacturer} is a superkey, but not a key (Why?).

If a relation has more than one keys, we can select any one (arbitrarily) to be the primary key. Primary Key attributes are underlined in the schema:

CAR(State, LicensePlateNo, VehicleID, Model, Year, Manufacturer)

C. Entity Integrity Constraints

The primary key attribute, PK, of any relational schema R in a database cannot have null values in any tuple. In other words, for each table in a DB, there must be a key; for each key, every row in the table must have non-null values. This is because PK is used to identify the individual tuples.

Mathematically, t[PK] != NULL for any tuple t € r(R).

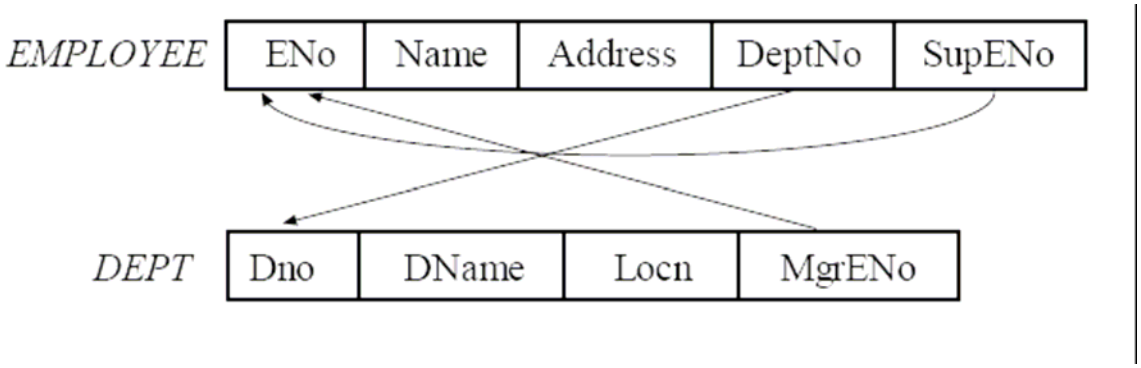
D. Referential Integrity Constraints

Referential integrity constraints are used to specify the relationships between two relations in a database.

Consider a referencing relation, R1, and a referenced relation, R2. Tuples in the referencing relation, R1, have attributed FK (called **foreign key** attributes) that reference

the primary key attributes of the referenced relation, R2. A tuple, t1, in R1 is said to reference a tuple, t2, in R2 if t1[FK] = t2[PK].

A referential integrity constraint can be displayed in a relational database schema as a directed arc from the referencing (foreign) key to the referenced (primary) key. Examples are shown in the figure below:



ER-to-Relational Mapping

Now we are ready to lay down some informal methods to help us create the Relational schemas from our ER models. These will be described in the following steps:

1. For each regular entity, E, in the ER model, create a relation R that includes all the simple attributes of E. Select the primary key for E, and mark it.
2. For each weak entity type, W, in the ER model, with the Owner entity type, E, create a relation R with all attributes of W as attributes of W, plus the primary key of E. [Note: if there are identical tuples in W which share the same owner tuple, then we need to create an additional index attribute in W.]
3. For each binary relation type, R, in the ER model, identify the participating entity types, S and T.
    - For 1:1 relationship between S and T

      Choose one relation, say S. Include the primary key of T as a foreign key of S.
    - For 1:N relationship between S and T

      Let S be the entity on the N side of the relationship. Include the primary key of T as a foreign key in S.
    - For M: N relation between S and T

Create a new relation, P, to represent R. Include the primary keys of both, S and T as foreign keys of P.

4. For each multi-valued attribute A, create a new relation, R, that includes all attributes corresponding to A, plus the primary key attribute, K, of the relation that represents the entity type/relationship type that has A as an attribute.
5. For each n-ary relationship type, n > 2, create a new relation S. Include as foreign key attributes in S the primary keys of the relations representing each of the participating entity types. Also include any simple attributes of the n-ary relationship type as attributes of S.

3.4 Relational Languages

We have so far considered the structure of a database; the relations and the associations between relations. In this section we consider how useful data may be extracted and filtered from database tables. A *relational language* is needed to express these *queries* in a well defined way. **A *relational language* is an abstract language which provides the database user with an interface through which they can specify data to be retrieved according to certain selection criteria**. The two main relational languages are relational algebra and relational calculus. Relational algebra, which we focus on here, provides the user with a set of operators which may be used to create new (temporary) relations based on information contained in existing relations. Relational calculus, on the other hand, provides a set of key words to allow the user to make ad hoc inquiries.

3.5 Relational Algebra

Relational algebra is a *procedural language* consisting of a set of *operators*. Each operator takes one or more relations as its input and produces *one* relation as its output. The seven basic relational algebra operations are **Selection**, **Projection**, **Joining**, **Union**, **Intersection**, **Difference** and **Division**. It is important to note that these operations do not alter the database. The relation produced by an operation is available to the user but it is not stored in the database by the operation.

Selection (also called *Restriction*)

The SELECT operator selects all tuples from some relation, so that some attributes in each tuple satisfy some condition. A new relation containing the selected tuples is then created as output. Suppose we have the relation STORES:

RELATION: STORES

Store-ID	Phone	Location	No-Bins
ST-A	6740394	Dublin	200
ST-B	8739404	Galway	310
ST-C	6372983	Dublin	75
ST-D	8393044	Cork	105

The relational operation:

R l = SELECT STORES WHERE Location = 'Dublin'

selects all tuples for stores that are located in Dublin and creates the new relation R1 which appears as follows:

RELATION: R1

Store-ID	Phone	Location	No-Bins
ST-A	6740394	Dublin	200
ST-C	6372983	Dublin	75

We can also impose conditions on more than one attribute. For example,

R2 = SELECT STORES WHERE Location = 'Dublin' AND No-Bins > 100

This operation selects only one tuple from the relation:

RELATION: R2

Store-ID	Phone	Location	No-Bins
ST-A	6740394	Dublin	200

Projection

The projection operator constructs a new relation from some existing relation by selecting only specified attributes of the existing relation and eliminating duplicate tuples in the newly formed relation. For example,

R3 = PROJECT STORES OVER Store-ID, Location

results in:

RELATION: R3

Store-ID	Location
ST-A	Dublin
ST-B	Galway
ST-C	Dublin
ST-D	Cork

Given the following operation,

R4 = PROJECT STORES OVER Location

What would the relation **R4** look like?

Joining

Joining is a operation for combining two relations into a single relation. At the outset, it requires choosing the attributes to match the tuples in each relation. Tuples in different relations but with the same value of matching attributes are combined into a single tuple in the output relation.

For example, with a new relation ITEMS:

RELATION: ITEMS

Item-No	Qty	Store-ID
I1	50	ST-A
I2	17	ST-A
I3	20	ST-B
I4	11	ST-C

… and our previous STORES relation:

RELATION: STORES

Store-ID	Phone	Location	No-Bins
ST-A	6740394	Dublin	200
ST-B	8739404	Galway	310
ST-C	6372983	Dublin	75
ST-D	8393044	Cork	105

if we joined ITEMS to STORES using the operator:

R5 = JOIN STORES, ITEMS OVER Store-ID

the resulting relation R5 would appear as follows:

RELATION: R5

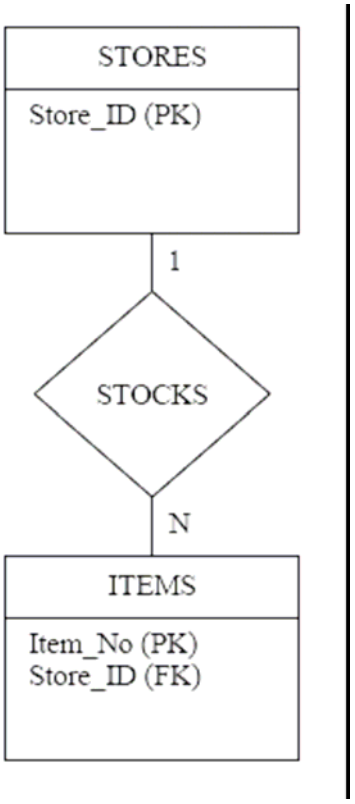
Store-ID	Phone	Location	No-Bins	Item-No	Qty
ST-A	6740394	Dublin	200	I1	50
ST-A	6740394	Dublin	200	I2	17
ST-B	8739404	Galway	310	I3	20
ST-C	6372983	Dublin	75	I4	11

This relation resulted from a joining of ITEMS and STORES over the common attribute Store-ID, i.e. any tuples of each relation which contained the same value of Store-ID were joined together to form a single tuple.

Joining relations together based on equality of values of common attributes is called an *equijoin*. Conditions of join may be other than equality - we may also have a 'greater-than' or 'less-than' join.

When duplicate attributes are removed from the result of an equijoin this is called an **natural join**. The example above is such a natural join - as Store-ID appears only once in the result.

Note that there is often a connection between keys (primary and foreign) and the attributes over which a join is performed in order to amalgamate information from multiple related tables in a database. In the above example, ITEMS.Store_ID is a foreign key reflecting the primary key STORE.Store_ID. When we join on Store_ID the relationship between the tables is expressed explicitly in the resulting output table. To illustrate, the relationship between these relations can be expressed as an E-R diagram, shown below.



Union, Intersection and Difference

These are the standard set operators. The requirement for carrying out any of these operations is that the two operand relations are **union-compatible** - i.e. they have the

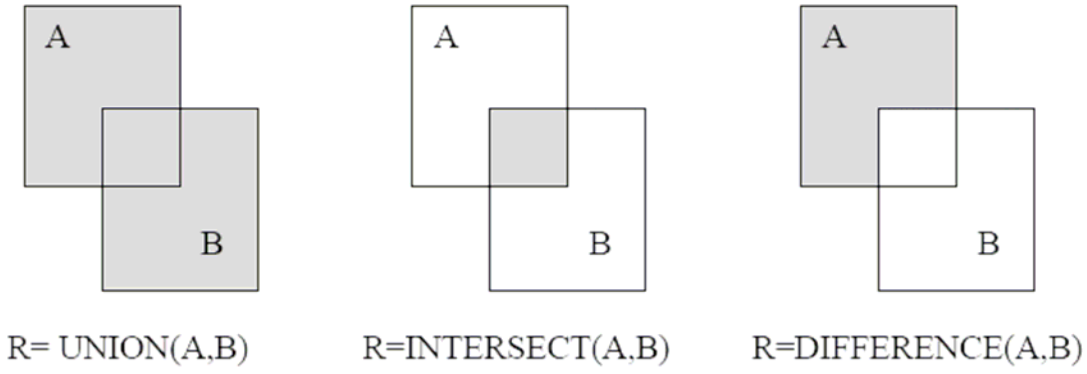
same number of attributes (say n) and the i^{th} attribute of each relation (i = l,…,n) must be from the same domain (they do not have to have the same attribute names).

The **UNION** operator builds a relation consisting of all tuples appearing in either or both of two specified relations.

The **INTERSECT** operator builds a relation consisting of all tuples appearing strictly in both specified relations.

The **DIFFERENCE** operator builds a relation consisting of all tuples appearing in the first, but not the second of two specified relations.

This may be represented diagrammatically as shown below.



As an exercise, find:

C = UNION(A,B), C = INTERSTION(A,B) and C= DIFFERENCE(A,B).

A

X	Y	Z
1	A	Red
2	B	Pink
3	C	Blue

B

X	Y	Z
1	A	Brown
1	A	Red
4	D	Black

Division

In its simplest form, this operation has a binary relation R(X,Y) as the dividend and a divisor that includes Y. The output is a set, *S*, , of values of X such that $x \in S$ if there is a row (*x*,*y*) in R for each *y* value in the divisor.

As an example, suppose we have two relations R6 and R7:

RELATION: R6

Store-ID	Location
C1	Boston
C2	Chicago
C1	Washinton
C3	Boston
C2	New York
C3	New York
C3	Chicago

RELATION: R7

Location
Boston
New York

The operation:

R8 = R6 / R7

will give the result:

RELATION: R8

Company
C3

This is because C3 is the only company for which there is a row with Boston and New York. The other companies, C1 and C2, do not satisfy this condition.

3.6 A Relational Database Management Systems-ORACLE

The Oracle database is a relational database system from Oracle corporation extensively used in product and internet-based applications in different platforms. Oracle database was developed by Larry Ellison, along with friends and former coworkers Bob Miner and Ed Oates, who had started a consultancy called Software Development Laboratories (SDL). They called their finished product Oracle, after the code name of a CIA-funded project they had worked on at a previous employer, Ampex.

Oracle9i Database Rel c 2 features full XML database functionality with Oracle XML DB, enhancements to the groundbreaking Oracle Real Application Clusters, and self-tuning and self-management capabilities to help improve DBA productivity and

efficiency. In addition, the built-in OLAP functionality has been expanded and significant enhancements and optimizations have been made for the Windows and Linux operating systems

Some of the Oracle database are as follows:

- Enterprise User Security -Password Based Enterprise User Security - Administering user accounts is a very time consuming and costly activity in many organizations. For example, users may lose their passwords, change roles or leave the company. Without timely user administration, the field is open for data misuse and data loss. By introducing password-based authentication, Oracle 9i Advanced Security has improved the ease-of-use and simplified enterprise user setup and administration.
- Oracle Partitioning -Oracle Partitioning, an option of Oracle9i Enterprise Edition, can enhance the manageability, performance, and availability of a wide variety of applications. Partitioning allows tables, indexes, and index-organized tables to be subdivided into smaller pieces, enabling these database objects to be managed and accessed at a finer level of granularity.
- Oracle Generic Connectivity and Oracle Transparent Gateway -Oracle offers two connectivity solutions to address the needs of disparate data access. They are: Oracle Generic Connectivity and Oracle Transparent Gateways. These two solutions make it possible to access any number of non-Oracle systems from an Oracle environment in a heterogeneously distributed environment.

- Performance Improvements -Performance is always a big issue with databases. The biggest improvements have been to Parallel Server which Oracle now calls Real Application Clusters and which allow applications to use clustered servers without modification.

- Security Enhancements -As the number of users increase and the locations and types of users become more diverse, better security (and privacy) features become essential.

3.7 Summary

1. To convert this ER into a **logical design** of a relational database. The logical model is also called a **Relational Model**.
2. **Domain**: a set of atomic values that an attribute can take
3. **Attribute**: name of a column in a particular table (all data is stored in tables).
4. The design of one table, containing the name of the table (i.e. the name of the relation), and the names of all the columns, or attributes is called **relational schema.**
5. The number of attributes in the relation's schema is called the degree of a relation.
6. A **super key** is a set of attributes whose values can be used to uniquely identify a tuple within a relation.
7. A **candidate key** is a super key that is minimal; that is, there is no proper subset that is itself a superkey.
8. The **primary key** of a relation is a candidate key especially selected to be the key for the relation.
9. The attribute(s) within one relation that matches a candidate key of another relation is called **foreign key(s).**
10. Each relational schema must satisfy the following four types of constraints viz. domain constraints, key constraints, entity integrity and referential integrity constraints.
11. **Relational algebra** is a *procedural language* consisting of a set of *operators*. Each operator takes one or more relations as its input and produces *one* relation as its output.
12. The seven basic relational algebra operations are **Selection**, **Projection**, **Joining**, **Union**, **Intersection**, **Difference** and **Division.**

3.8 Key Words

Relation, Integrity Constraints, Relational Algebra, Primary Key, Foreign Key, Super Key, Candidate Key, Degree of a Relation, Projection, Join

3.9 Self assessment Questions

1. Explain in brief the relational approach to data base structures.
2. What is a relation? What are its characteristics?
3. Explain any three relational operators with example.
4. Explain various relational constraints with example.
5. Explain Relational Algebra.What are the relational algebra operations that can be performed?
6. What are the advantages of Relational Model?

3.10 References/Suggested Readings

1. Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000
2. Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld
3. Elamasri R . and Navathe, S., Fundamentals of Database Systems (3rd Edition), Pearsson Education, 2000.

MCA 202/MS 11

Author: Abhishek Taneja **Vetter: Prof. Dharminder Kumar**
Lesson: SQL **Lesson No. : 04**

Structure

4.0 Objectives

At the end of this chapter the reader will be able to:

- Describe history of SQL
- Describe various SQL commands
- Define characteristics of SQL commands
- Describe DDL,DML and DCL commands
- Describe constraints and indexes in SQL

4.1 Introduction and History

In this chapter we want to emphasize that SQL is both deep and wide. Deep in the sense that it is implemented at many levels of database communication, from a simple Access form list box right up to high-volume communications between mainframes. SQL is widely implemented in that almost every DBMS supports SQL statements for communication. The reason for this level of acceptance is partially explained by the amount of effort that went into the theory and development of the standards.

Current State

So the ANSI-SQL group has published three standards over the years:

- SQL89 (SQL1)
- SQL92 (SQL2)
- SQL99 (SQL3)

The vast majority of the language has not changed through these updates. We can all profit from the fact that almost all of the code we wrote to SQL standards of 1989 is still perfectly usable. Or in other words, as a new student of SQL there is over ten years of SQL code out there that needs your expertise to maintain and expand. 1

Most DBMS are designed to meet the SQL92 standard. Virtually all of the material in this book was available in the earlier standards as well. Since many of the advanced features of SQL92 have yet to be implemented by DBMS vendors, there has been little pressure for a new version of the standard. Nevertheless a SQL99 standard was developed to address advanced issues in SQL. All of the core functions of SQL, such as adding, reading and modifying data, are the same. Therefore, the topics in this book are not affected by the new standard. As of early 2001, no vendor has implemented the SQL99 standard.

There are three areas where there is current development in SQL standards. First entails improving Internet access to data, particularly to meet the needs of the emerging XML standards. Second is integration with Java, either through Sun's Java Database

Connectivity (JDBC) or through internal implementations. Last, the groups that establish SQL standards are considering how to integrate object- based programming models.

4.2 What is SQL?

Structured Query Language, commonly abbreviated to SQL and pronounced as "sequel", is not a conventional computer programming language in the normal sense of the phrase. It allows users to access data in relational database management systems. SQL is about data and results, each SQL statement returns a result, whether that result be a query, an update to a record or the creation of a database table. SQL is most often used to address a relational database, which is what some people refer to as a SQL database.So in brief we can describe SQL as follows:

- SQL stands for Structured Query Language
- SQL allows you to access a database
- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert new records in a database
- SQL can delete records from a database
- SQL can update records in a database
- SQL is easy to learn

Creating a Database

Many database systems have graphical interfaces which allow developers (and users) to create, modify and otherwise interact with the underlying database management system (DBMS). However, for the purposes of this chapter all interactions with the DBMS will be via SQL commands rather than via menus.

4.3 SQL Commands

There are three groups of commands in SQL:

1. Data Definition

2. Data Manipulation and

3. Transaction Control

4.3.1 Characteristics of SQL Commands

Here you can see that SQL commands follow a number of basic rules:

• SQL keywords are not normally case sensitive, though this in this tutorial all commands (SELECT, UPDATE etc) are upper-cased.

• Variable and parameter names are displayed here as lower-case.

• New-line characters are ignored in SQL, so a command may be all on one line or broken up across a number of lines for the sake of clarity.

• Many DBMS systems expect to have SQL commands terminated with a semi-colon character.

4.4 Data Definition Language (DDL) in SQL

The Data Definition Language (DDL) part of SQL permits database tables to be created or deleted. We can also define indexes (keys), specify links between tables, and impose constraints between database tables.

The most important DDL statements in SQL are:

- CREATE TABLE - creates a new database table
- ALTER TABLE - alters (changes) a database table
- DROP TABLE - deletes a database table

How to create table

Creating a database is remarkably straightforward. The SQL command which you have to give is just:

CREATE DATABASE dbname;

In this example you will call the database GJUniv, so the command which you have to give is:

CREATE DATABASE GJUniv;

Once the database is created it, is possible to start implementing the design sketched out previously.

So you have created the database and now it's time to use some SQL to create the tables required by the design. Note that all SQL keywords are shown in upper case, variable names in a mixture of upper and lower case.

The SQL statement to create a table has the basic form:

CREATE TABLE name(col1 datatype, col2 datatype, …);

So, to create our User table we enter the following command:

CREATE TABLE User (FirstName TEXT, LastName TEXT, UserID TEXT, Dept TEXT, EmpNo INTEGER, PCType TEXT);

The TEXT datatype, supported by many of the most common DBMS, specifies a string of characters of any length. In practice there is often a default string length which varies by product. In some DBMS TEXT is not supported, and instead a specific string length has to be declared. Fixed length strings are often called CHAR(x), VCHAR(x) or VARCHAR(x), where x is the string length. In the case of INTEGER there are often multiple flavors of integer available. Remembering that larger integers require more bytes for data storage, the choice of int size is usually a design decision that ought to be made up front.

How to Modify table

Once a table is created it's structure is not necessarily fixed in stone. In time requirements change and the structure of the database is likely to evolve to match your wishes. SQL can be used to change the structure of a table, so, for example, if we need to add a new field to our User table to tell us if the user has Internet access, then we can execute an SQL ALTER TABLE command as shown below:

ALTER TABLE User ADD COLUMN Internet BOOLEAN;

To delete a column the ADD keyword is replaced with DROP, so to delete the field we have just added the SQL is:

ALTER TABLE User DROP COLUMN Internet;

How to delete table

If you have already executed the original CREATE TABLE command your database will already contain a table called User, so let's get rid of that using the DROP command:

DROP TABLE User;

And now we'll recreate the User table we'll use throughout the rest of this tutorial:

CREATE TABLE User (FirstName VARCHAR (20), LastName VARCHAR (20), UserID VARCHAR(12) UNIQUE, Dept VARCHAR(20), EmpNo INTEGER UNIQUE, PCType VARCHAR(20);

4.5 Data Manipulation Language in SQL (DML)

SQL language also includes syntax to update, insert, and delete records. These query and update commands together form the Data Manipulation Language (DML) part of SQL:

- INSERT INTO - inserts new data into a database table
- UPDATE - updates data in a database table
- DELETE - deletes data from a database table
- SELECT - extracts data from a database table

How to Insert Data

Having now built the structure of the database it is time to populate the tables with some data. In the vast majority of desktop database applications data entry is performed via a user interface built around some kind of GUI form. The form gives a representation of the information required for the application, rather than providing a simple mapping onto the tables. So, in this sample application you would imagine a form with text boxes for the user details, drop-down lists to select from the PC table, drop-down selection of the software packages etc. In such a situation the database user is shielded both from the underlying structure of the database and from the SQL which may be used to enter data into it. However we are going to use the SQL directly to populate the tables so that we can move on to the next stage of learning SQL.

The command to add new records to a table (usually referred to as an append query), is:

INSERT INTO target [(field1[, field2[, ...]])]
VALUES (value1[, value2[, ...]);

So, to add a User record for user Jim Jones, we would issue the following INSERT query:

INSERT INTO User (FirstName, LastName, UserID, Dept, EmpNo, PCType) 6
VALUES ("Jim", "Jones", "Jjones","Finance", 9, "DellDimR450");

Obviously populating a database by issuing such a series of SQL commands is both tedious and prone to error, which is another reason why database applications have front-ends. Even without a specifically designed front-end, many database systems - including MS Access - allow data entry direct into tables via a spreadsheet-like interface.

The INSERT command can also be used to copy data from one table into another. For example, The SQL query to perform this is:

INSERT INTO User (FirstName, LastName, UserID, Dept, EmpNo, PCType, Internet)
SELECT FirstName, LastName, UserID, Dept, EmpNo, PCType, Internet
FROM NewUsers;

How to Update Data

The INSERT command is used to add records to a table, but what if you need to make an amendment to a particular record? In this case the SQL command to perform updates is the UPDATE command, with syntax:

UPDATE table
SET newvalue
WHERE criteria;

For example, let's assume that we want to move user Jim Jones from the Finance department to Marketing. Our SQL statement would then be:

UPDATE User
SET Dept="Marketing"
WHERE EmpNo=9;

Notice that we used the EmpNo field to set the criteria because we know it is unique. If we'd used another field, for example LastName, we might have accidentally updated the records for any other user with the same surname.

The UPDATE command can be used for more than just changing a single field or record at a time. The SET keyword can be used to set new values for a number of different fields, so we could have moved Jim Jones from Finance to marketing and changed the PCType as well in the same statement (SET Dept="Marketing", PCType="PrettyPC"). Or if all of the Finance department were suddenly granted Internet access then we could have issued the following SQL query:

```
UPDATE User
SET Internet=TRUE
WHERE Dept="Finance";
```

You can also use the SET keyword to perform arithmetical or logical operations on the values. For example if you have a table of salaries and you want to give everybody a 10% increase you can issue the following command:

```
UPDATE PayRoll
SET Salary=Salary * 1.1;
```

How to Delete Data

Now that we know how to add new records and to update existing records it only remains to learn how to delete records before we move on to look at how we search through and collate data. As you would expect SQL provides a simple command to delete complete records. The syntax of the command is:

```
DELETE [table.*]
FROM table
WHERE criteria;
```

Let's assume we have a user record for John Doe, (with an employee number of 99), which we want to remove from our User we could issue the following query:

```
DELETE *
FROM User
```

WHERE EmpNo=99;

In practice delete operations are not handled by manually keying in SQL queries, but are likely to be generated from a front end system which will handle warnings and add safe-guards against accidental deletion of records.

Note that the DELETE query will delete an entire record or group of records. If you want to delete a single field or group of fields without destroying that record then use an UPDATE query and set the fields to Null to over-write the data that needs deleting. It is also worth noting that the DELETE query does not do anything to the structure of the table itself, it deletes data only. To delete a table, or part of a table, then you have to use the DROP clause of an ALTER TABLE query.

4.6 Transaction Control Language in SQL(TCL)

The SQL Data Control Language (DCL) provides security for your database. The DCL consists of the GRANT, REVOKE, COMMIT, and ROLLBACK statements. GRANT and REVOKE statements enable you to determine whether a user can view, modify, add, or delete database information.

Working with transaction control

Applications execute a SQL statement or group of logically related SQL statements to perform a database transaction. The SQL statement or statements add, delete, or modify data in the database.

Transactions are atomic and durable. To be considered atomic, a transaction must successfully complete all of its statements; otherwise none of the statements execute. To be considered durable, a transaction's changes to a database must be permanent.

Complete a transaction by using either the COMMIT or ROLLBACK statements. COMMIT statements make permanent the changes to the database created by a transaction. ROLLBACK restores the database to the state it was in before the transaction was performed.

SQL Transaction Control Language Commands (TCL.)

This page contains some SQL TCL. commands that I think it might be useful. Each command's description is taken and modified from the SQLPlus help. They are provided as is and most likely are partially described. So, if you want more detail or other commands, please use HELP in the SQLPlus directly.

COMMIT

PURPOSE:

To end your current transaction and make permanent all changes performed in the transaction. This command also erases all savepoints in the transaction and releases the transaction's locks. You can also use this command to manually commit an in-doubt distributed transaction.

SYNTAX:

COMMIT [WORK]
[COMMENT 'text'
| FORCE 'text' [, integer]]

Where:

- WORK : is supported only for compliance with standard SQL. The statements COMMIT and COMMIT WORK are equivalent.
- COMMENT : specifies a comment to be associated with the current transaction. The 'text' is a quoted literal of up to 50 characters that Oracle stores in the data dictionary view DBA_2PC_PENDING along with the transaction ID if the transaction becomes in-doubt.
- FORCE : manually commits an in-doubt distributed transaction. The transaction is identified by the 'text' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. You can also use the integer to specifically assign the transaction a system change number (SCN). If you omit the integer, the transaction is committed using the current SCN.

COMMIT statements using the FORCE clause are not supported in PL/SQL.

PREREQUISITES:

You need no privileges to commit your current transaction. To manually commit a distributed in-doubt transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually commit a distributed in-doubt transaction that was originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

Example:

To commit your current transaction, enter

SQL> **COMMIT WORK**;
Commit complete.

ROLLBACK

PURPOSE:

To undo work done in the current transaction. You can also use this command to manually undo the work done by an in-doubt distributed transaction.

SYNTAX:

ROLLBACK [WORK]
[TO [SAVEPOINT] savepoint
| FORCE 'text']

Where:

- WORK : is optional and is provided for ANSI compatibility.
- TO : rolls back the current transaction to the specified savepoint. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.
- FORCE : manually rolls back an in-doubt distributed transaction. The transaction is identified by the 'text' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. ROLLBACK statements with the FORCE clause are not supported in PL/SQL.

PREREQUISITES: To roll back your current transaction, no privileges are necessary. To manually roll back an in-doubt distributed transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually roll back an in-doubt distributed transaction originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

Example:

To rollback your current transaction, enter

SQL> **ROLLBACK**;
Rollback complete.

Creating users

This section covers the following information:

- Creating database administrators
- Creating users

The CREATE statement is not a part of the Data Control Language, but rather the Data Definition Language. This chapter addresses the CREATE statement as it relates to the creation of database administrators and users.

Creating database administrators

Database security is defined and controlled by database administrators (DBAs). Within the scope of database security, DBAs are responsible for:

- Adding users.
- Deleting users.
- Permitting access to specific database objects.
- Limiting or prohibiting access to database objects.

- Granting users privileges to view or modify database objects.
- Modifying or revoking privileges that have been granted to the users.

A user who initially creates a database becomes its default administrator. Therefore, this initial user has the authority to create other administrator accounts for that particular database. OpenEdge Studio offers two methods for creating DBAs:

- In SQL, the DBA uses the CREATE statement to create a user and then uses the GRANT statement to provide the user with administrative privileges.
- In Progress 4GL, a DBA uses the Data Administration Tool to create other administrators.

Creating users

Use the following syntax to employ the CREATE USER statement:

Syntax

CREATE USER 'username', 'password' ;

In **Example 4-1**, an account with DBA privileges creates the 'username' 'GPS' with 'password' 'star'.

Example 4-1: CREATE USER statement

CREATE USER 'GPS', 'star';

A user's password can be changed easily by using the ALTER USER statement:

Syntax

ALTER USER 'username', 'old_password', 'new_password';

Example 4-2 demonstrates the use of the ALTER USER statement.

Example 4-2: ALTER USER statement

ALTER USER 'GPS', 'star', 'star1';

- When users are created, the default DBA (the user who created the database) becomes disabled. It is important to grant DBA access to at least one user so you will have a valid DBA account.
- The user's password can be easily changed using the ALTER USER statement.

Granting privileges

This section covers the following information:

- Privilege basics
- GRANT statement

Privilege basics

There are two types of privileges¾those granted on databases and those granted on tables, views, and procedures.

Privileges for databases:

- Granting or restricting system administration privileges (DBA).
- Granting or restricting general creation privileges on a database (RESOURCE).

Privileges granted on tables, views, and procedures grant or restrict operations on specific operations, such as:

- Altering an object definition.
- Deleting, inserting, selecting and updating records.
- Executing stored procedures.
- Granting privileges.
- Defining constraints to an existing table.

GRANT statement

The GRANT statement can be used to provide the user with two different types of privileges:

- **Database-wide privileges**
- **Table-specific privileges**

Database-wide privileges

Database-wide privileges grant the user either DBA or RESOURCE privileges. Users with DBA privileges have the ability to access, modify, or delete a database object and to grant privileges to other users. RESOURCE privileges allow a user to create database objects.

The GRANT statement syntax for granting RESOURCE or DBA privileges is:

Syntax

GRANT {RESOURCE, DBA }

TO username [, username], ... ;

The following statement provides resource privileges to user 'GSP'.

Example 4-3: GRANT RESOURCE statement

GRANT RESOURCE TO 'GSP';

In this case, GSP is granted the privilege to issue CREATE statements, and can therefore add objects, such as tables, to the database.

Table-specific privileges can be granted to users so they can view, add, delete, or create indexes for data within a table. Privileges can also be granted to allow users to refer to a table from another table's constraint definitions.

The GRANT statement syntax for granting table-specific privileges is:

Syntax

GRANT {privilege [, privilege], ... |ALL}

ON table_name

TO {username [, username], ... | PUBLIC}

[WITH GRANT OPTION] ;

This is the syntax for the privilege value:

Syntax

{ SELECT | INSERT | DELETE | INDEX

| UPDATE [(column , column , ...)]

| REFERENCES [(column , column , ...)] }

In this instance, a DBA restricts the types of activities a user is allowed to perform on a table. In the following example, 'GSP' is given permission to update the item name, item number, and catalog descriptions found in the item table.

Note: By employing the WITH GRANT OPTION clause, you enable a user to grant the same privilege he or she has been granted to others. This clause should be used carefully due to its ability to affect database security.

Example 4-4 illustrates the granting of table-specific privileges:

Example 4-4: GRANT UPDATE statement

GRANT UPDATE

ON Item (ItemNum, ItemName, CatDescription)

TO 'GSP';

The GRANT UPDATE statement has limited GSP's ability to interact with the item table. Now, if GSP attempts to update a column to which he has not been granted access, the database will return the error message in **Example 4-5:**

Example 4-5: SQL error message

=== SQL Exception 1 ===

SQLState=HY000

ErrorCode=-20228

[JDBC Progress Driver}:Access Denied (Authorisation

failed) (7512)

Granting public access

The GRANT statement can be easily modified to make previously restricted columns accessible to the public, as in Example 4-6.

Example 4-6: Granting update privilege to public

GRANT UPDATE

ON Item (ItemNum, ItemName, CatDescription)

TO PUBLIC;

Revoking privileges

The REVOKE statement can be used for a wide variety of purposes. It can revoke a single user's access to a single column or it can revoke the public's privilege to access an entire database.

Privileges are revoked in the same manner in which they are granted_database-wide or table-specific.

The syntax for using the REVOKE statement to revoke database-wide privileges is:

Syntax

REVOKE {RESOURCE, DBA}

FROM {username [, username], ...};

The syntax for using the REVOKE statement to revoke table-specific privileges is:

Syntax

REVOKE [GRANT OPTION FOR] {privilege [, privilege], ... |ALL

[PRIVILEGES]} ON table_name

FROM {username[,username], ... |PUBLIC} [RESTRICT|CASCADE];

where privilege is:

{EXECUTE|SELECT|INSERT|DELETE|INDEX|UPDATE [(COLUMN, COLUMN, ...)]|

REFERENCES [(COLUMN, COLUMN, ...)]}

The REVOKE statement can be used to remit the privileges previously granted to 'GPS' as shown in **Example 4-7**.

Example 4-7: REVOKE statement

REVOKE UPDATE

ON Item (ItemNum, ItemName, CatDescription)

FROM "GPS"

If the REVOKE statement specifies RESTRICT, SQL checks if the privilege being revoked was passed on to other users. This is possible only if the original privilege included the WITH GRANT OPTION clause. If so, the REVOKE statement fails and generates an error. If the privilege was not passed on, the REVOKE statement succeeds.

If the REVOKE statement specifies CASCADE, revoking the access privileges from a user also revokes the privileges from all users who received the privilege from that user.

If the REVOKE statement specifies neither RESTRICT nor CASCADE, the behavior is the same as for CASCADE.

4.7 Constraints in SQL

Data types are a way to limit the kind of data that can be stored in a table. For many applications, however, the constraint they provide is too coarse. For example, a column containing a product price should probably only accept positive values. But there is no data type that accepts only positive numbers. Another issue is that you might want to constrain column data with respect to other columns or rows. For example, in a table containing product information, there should only be one row for each product number.

To that end, SQL allows you to define constraints on columns and tables. Constraints give you as much control over the data in your tables as you wish. If a user attempts to store data in a column that would violate a constraint, an error is raised. This applies even if the value came from the default value definition.

4.7.1 Check Constraints

A check constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy an arbitrary expression. For instance, to require positive product prices, you could use:

CREATE TABLE products (product_no integer, name text, price numeric CHECK (price > 0));

As you see, the constraint definition comes after the data type, just like default value definitions. Default values and constraints can be listed in any order. A check constraint consists of the key word *CHECK* followed by an expression in parentheses. The check constraint expression should involve the column thus constrained, otherwise the constraint would not make too much sense.

4.7.2 Not-Null Constraints

A not-null constraint simply specifies that a column must not assume the null value. A syntax example:

CREATE TABLE products (product_no integer NOT NULL, name text NOT NULL, price numeric);

A not-null constraint is always written as a column constraint. A not-null constraint is functionally equivalent to creating a check constraint CHECK (column_name IS NOT NULL), but in PostgreSQL creating an explicit not-null constraint is more efficient. The drawback is that you cannot give explicit names to not-null constraints created that way.

4.7.3 Unique Constraints

Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table. The syntax is

CREATE TABLE products (product_no integer UNIQUE, name text, price numeric);

when written as a column constraint, and

CREATE TABLE products (product_no integer, name text, price numeric,UNIQUE (product_no));

when written as a table constraint.

4.7.4 Primary Key Constraints

Technically, a primary key constraint is simply a combination of a unique constraint and a not-null constraint. So, the following two table definitions accept the same data:

CREATE TABLE products (product_no integer UNIQUE NOT NULL, name text, price numeric);

CREATE TABLE products (product_no integer PRIMARY KEY,name text, price numeric);

Primary keys can also constrain more than one column; the syntax is similar to unique constraints:

CREATE TABLE example (a integer,b integer,c integer, PRIMARY KEY (a, c));

A primary key indicates that a column or group of columns can be used as a unique identifier for rows in the table. (This is a direct consequence of the definition of a primary key. Note that a unique constraint does not, in fact, provide a unique identifier because it does not exclude null values.) This is useful both for documentation purposes and for

client applications. For example, a GUI application that allows modifying row values probably needs to know the primary key of a table to be able to identify rows uniquely.

4.7.5 Foreign Keys Constraints

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the referential integrity between two related tables.

Say you have the product table that we have used several times already:

```
CREATE TABLE products (product_no integer PRIMARY KEY, name text,
    price numeric);
```

Let's also assume you have a table storing orders of those products. We want to ensure that the orders table only contains orders of products that actually exist. So we define a foreign key constraint in the orders table that references the products table:

```
CREATE TABLE orders ( order_id integer PRIMARY KEY,product_no integer
REFERENCES products (product_no), quantity integer);
```

Now it is impossible to create orders with product_no entries that do not appear in the products table.

We say that in this situation the orders table is the referencing table and the products table is the referenced table. Similarly, there are referencing and referenced columns.

4.8 Indexes in SQL

Relational databases like SQL Server use indexes to find data quickly when a query is processed. Creating and removing indexes from a database schema will rarely result in changes to an application's code; indexes operate 'behind the scenes' in support of the database engine. However, creating the proper index can drastically increase the performance of an application.

The SQL Server engine uses an index in much the same way a reader uses a book index. For example, one way to find all references to INSERT statements in a SQL book would be to begin on page one and scan each page of the book. We could mark each time we find the word INSERT until we reach the end of the book. This approach is pretty time consuming and laborious. Alternately, we can also use the index in the back of the book

to find a page number for each occurrence of the INSERT statements. This approach produces the same results as above, but with tremendous savings in time.

When a SQL Server has no index to use for searching, the result is similar to the reader who looks at every page in a book to find a word: the SQL engine needs to visit every row in a table. In database terminology we call this behavior a table scan, or just scan.

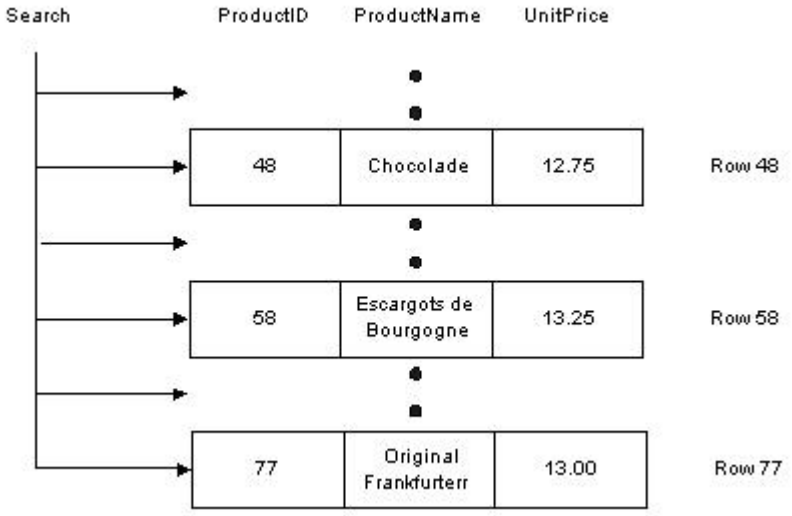
A table scan is not always a problem, and is sometimes unavoidable. However, as a table grows to thousands of rows and then millions of rows and beyond, scans become correspondingly slower and more expensive.

Consider the following query on the Products table of the Northwind database. This query retrieves products in a specific price range.

SELECT ProductID, ProductName, UnitPrice

FROM Products WHERE (UnitPrice > 12.5) AND (UnitPrice < 14)

There is currently no index on the Product table to help this query, so the database engine performs a scan and examines each record to see if UnitPrice falls between 12.5 and 14. In the diagram below, the database search touches a total of 77 records to find just three matches.



Now imagine if we created an index, just like a book index, on the data in the UnitPrice column. Each index entry would contain a copy of the UnitPrice value for a row, and a reference (just like a page number) to the row where the value originated. SQL will sort these index entries into ascending order. The index will allow the database to quickly

narrow in on the three rows to satisfy the query, and avoid scanning every row in the table.

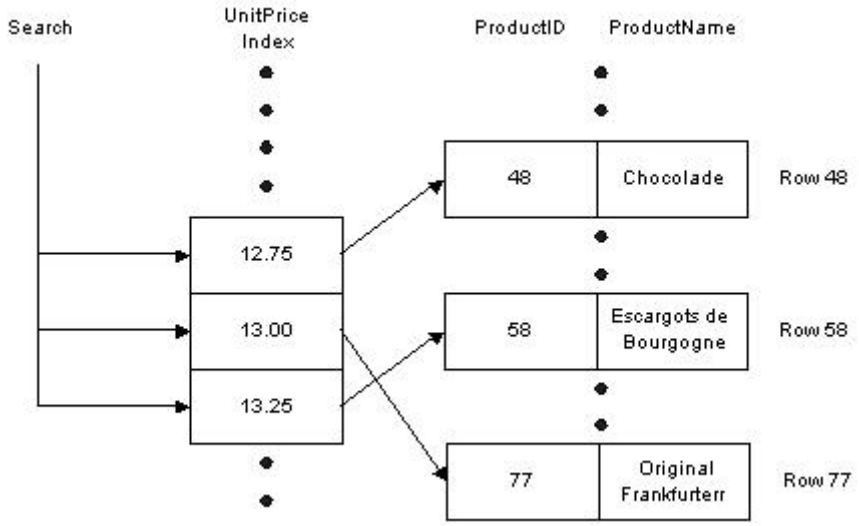
We can create the same index using the following SQL. The command specifies the name of the index (IDX_UnitPrice), the table name (Products), and the column to index (UnitPrice).

CREATE INDEX [IDX_UnitPrice] ON Products (UnitPrice)

How It Works

The database takes the columns specified in a CREATE INDEX command and sorts the values into a special data structure known as a B-tree. A B-tree structure supports fast searches with a minimum amount of disk reads, allowing the database engine to quickly find the starting and stopping points for the query we are using.

Conceptually, we may think of an index as shown in the diagram below. On the left, each index entry contains the index key (UnitPrice). Each entry also includes a reference (which points) to the table rows which share that particular value and from which we can retrieve the required information.



Much like the index in the back of a book helps us to find keywords quickly, so the database is able to quickly narrow the number of records it must examine to a minimum by using the sorted list of UnitPrice values stored in the index. We have avoided a table

scan to fetch the query results. Given this sketch of how indexes work, lets examine some of the scenarios where indexes offer a benefit.

4.9 Summary

1. SQL allows users to access data in relational database management systems
2. There are three groups of SQL commands viz., DDL, DML and DCL.
3. The **Data Definition Language** (DDL) part of SQL permits database tables to be created or deleted.
4. SQL language also includes syntax to update, insert, and delete records. These query and update commands together form the **Data Manipulation Language** (DML) part of SQL.
5. The SQL **Data Control Language** (DCL) provides security for your database. The DCL consists of the GRANT, REVOKE, COMMIT, and ROLLBACK statements.
6. **Constraints** are a way to limit the kind of data that can be stored in a table.
7. Relational databases like SQL Server use **indexes** to find data quickly when a query is processed.

4.10 Key Words

SQL, DDL, DML, DCL, Constraints, Indexes

4.11 Self Assessment Questions

1)What does SQL stand for?

2) What SQL statement is used to delete table “Student”?

3) How can you insert a new record in table “Department”

4) With SQL, how can you insert "GJU" as the "FName" in the "University" table?

5) How can you delete a record from table “student” where “RollNo”=GJU501?

6) Explain the use of Grant And Revoke Commands?

7) What are Transaction Control Language Commands?

8) Explain the ways to create a new user?

4.12 References/Suggested Readings

1. www.microsoft.com
2. Sams Teach Yourself Microsoft SQL Server 2000 in 21 Days by Richard Waymire, Rick Sawtell

3. Microsoft SQL Server 7.0 Programming Unleashed by John Papa, et al 20
4. Microsoft Sql Server 7.0 Resource Guide by Microsoft Corporation
5. http://www.cs.rpi.edu/~temtanay/dbf-fall97/oracle/sql_ddcmd.html

MCA 202/MS 11

Author: Abhishek Taneja **Vetter: Dr. Pradeep Bhatia**

Lesson: Relational Database Design and Normalization **Lesson No. : 05**

Structure

5.0 Objectives

At the end of this chapter the reader will be able to:

- Describe benefits of relational model
- Describe informal guidelines for relational schema
- Describe update, insertion and deletion anomalies
- Describe Functional Dependencies and its various forms
- Describe fundamentals of normalization
- Describe and distinguish the 1NF, 2NF, 3NF and BCNF normal forms.

- Describe Lossless join and Dependency preserving decomposition

5.1 Introduction

When designing a database, you have to make decisions regarding how best to take some system in the real world and model it in a database. This consists of deciding which tables to create, what columns they will contain, as well as the relationships between the tables. While it would be nice if this process was totally intuitive and obvious, or even better automated, this is simply not the case. A well-designed database takes time and effort to conceive, build and refine.

The benefits of a database that has been designed according to the relational model are numerous. Some of them are:

- Data entry, updates and deletions will be efficient.
- Data retrieval, summarization and reporting will also be efficient.
- Since the database follows a well-formulated model, it behaves predictably.
- Since much of the information is stored in the database rather than in the application, the database is somewhat self-documenting.
- Changes to the database schema are easy to make.

The objective of this chapter is to explain the basic principles behind relational database design and demonstrate how to apply these principles when designing a database.

5.2 Informal Design Guidelines for Relational Schemas

We discuss four *informal measures* of quality for relation schema design in this section:

- Semantics of the attributes
- Reducing the redundant values in tuples
- Reducing the null values in tuples
- Disallowing the possibility of generating spurious tuples

These measures are not always independent of one another, as we shall see.

5.2.1 Semantics of the attributes

The semantics, specifies how to interpret the attribute values stored in a tuple of the relation-in other words, how the attribute values in a tuple relate to one another. If the conceptual design is done carefully, followed by a systematic mapping into relations, most of the semantics will have been accounted for and the resulting design should have a clear meaning.

Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation. Intuitively, if a relation schema corresponds to one entity type or one relation

5.2.2 Reducing the redundant values in tuples

Storing the Same information redundantly, that is, in more than one place within a database, can lead to several problems:

Redundant Storage: Some information is stored repeatedly.

Update Anomalies: If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.

Insertion Anomalies: It may not be possible to store certain information unless some other, unrelated, information is stored as well.

Deletion Anomalies: It may not be possible to delete certain information without losing some other, unrelated, information as well.

Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly.

	name	*lot*	*rating*	*hourly_wages*	*hours_worked*
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Sruiley	22	8	10	30
131-24-3650	Srllethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Figure 5.1 An Instance of Hourly_Emps Relation

5.2.3 Reducing the null values in tuples

It is worth considering whether the use of *null* values can address some of these problems. As we will see in the context of our example, they cannot provide a complete solution, but they can provide some help. In this chapter, we do not discuss the use of *null* values beyond this one example. Consider the example Hourly_Elmps relation. Clearly, *null* values cannot help eliminate redundant storage or update anomalies. It appears that they can address insertion and deletion anomalies. For instance, to deal with the insertion anomaly example, we can insert an employee tuple with *null* values in the hourly wage field. However, *null* values cannot address all insertion anomalies. For example, we cannot record the hourly wage for a rating unless there is an employee with that rating, because we cannot store a null value in the *ssn* field, which is a primary key field. Similarly, to deal with the deletion anomaly examnple, we might consider storing a tuple with *null* values in all fields except *rating* and *hourly_wages* if the last tuple with a given *rating* would otherwise be deleted. However, this solution does not work because it requires the *8871,* value to be *null,* and primary key fields cannot be *null.* Thus, null values do not provide a general solution to the problems of redundancy, even though they can help in some cases.

Decompositions

Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural. Functional dependencies can 'be used to identify such situations and suggest refinements to the schema. The essential idea is that many problems arising from redundancy can be addressed by replacing a relation 'with a collection of smaller relations. A. decomposition of a relation schema onsists of replacing the relation schema by two (or more) relation schema that each contain a subset of the attributes of *R* and together include all attributes in *R.* Intuitively, we want to store the information in any given instance of *R* by storing projections of the instance. This section examines the use of decompositions through several examples. we can decompose Hourly_Emps into two relations:

Hourly_Emps2(ssn,name,lot,rating_hours_*worked*)

Wages (*rating, hourly_wages)*

The instances of these relations are shown in Figure 5.2 corresponding to instance shown in figure 5.1.

ssn	name	lot	rating	hours_worked
123-22-3666	Attishoo	48	8	40
231-31-5368	Sluiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

rating	hourly_wages
8	10
5	7

Figure 5.2 Instance of Hourly_Emps2 and Wages

Note that we can easily record the hourly wage for any rating simply by adding a tuple to Wages, even if no employee with that rating appears in the current instance of Hourly_Emps. Changing the wage associated with a rating involves updating a single Wages tuple. This is more efficient than updating several tuples (as in the original design), and it eliminates the potential for inconsistency.

5.2.4 Disallowing the possibility of generating spurious tuples

Design relation schemas so that they can be joined with equality conditions on attributes that are either primary keys or foreign keys in a way that guarantees that no spurious tuples are generated. Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations, because joining on such attributes may produce spurious tuples.

5.3 Functional Dependencies

For our discussion on functional dependencies assume that a relational schema has attributes (A, B, C... Z) and that the whole database is described by a single universal relation called R = (A, B, C, ..., Z). This assumption means that every attribute in the database has a unique name.

A functional dependency is a property of the semantics of the attributes in a relation. The semantics indicate how attributes relate to one another, and specify the functional dependencies between attributes. When a functional dependency is present, the dependency is specified as a constraint between the attributes.

Consider a relation with attributes A and B, where attribute B is functionally dependent on attribute A. If we know the value of A and we examine the relation that holds this dependency, we will find only one value of B in all of the tuples that have a given value of A, at any moment in time. Note however, that for a given value of B there may be several different values of A.

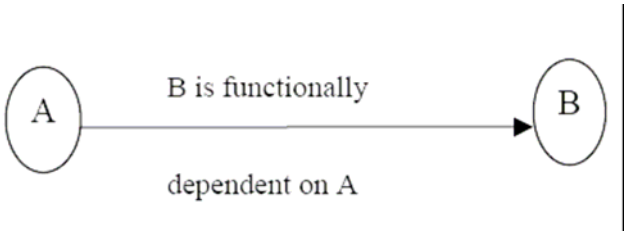


Figure 5.3

In the figure5.3 above, A is the determinant of B and B is the consequent of A.

The determinant of a functional dependency is the attribute or group of attributes on the left-hand side of the arrow in the functional dependency. The consequent of a fd is the attribute or group of attributes on the right-hand side of the arrow.

Identifying Functional Dependencies

Now let us consider the following Relational schema shown in figure 5.4.

STAFFBRANCH

staff#	sname	position	salary	branch#	baddress
SL21	Kristy	manager	30000	B005	22Deer Road
SG37	Debris	assistant	12000	B003	162Main Street
SG14	Alan	supervisor	18000	B003	163Main Street
SA9	Traci	assistant	12000	B007	375Fox Avenue
SG5	David	manager	24000	B003	163Main Street

Figure 5.4 Relational Schema

The functional dependency staff# → position clearly holds on this relation instance. However, the reverse functional dependency position → staff# clearly does not hold.

The relationship between staff# and position is 1:1 – for each staff member there is only one position. On the other hand, the relationship between position and staff# is 1:M – there are several staff numbers associated with a given position.

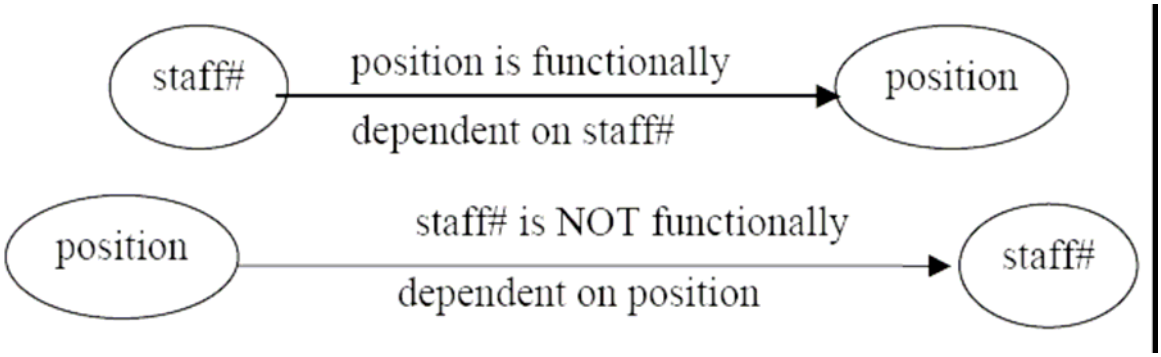


Figure 5.5

For the purposes of normalization we are interested in identifying functional dependencies between attributes of a relation that have a 1:1 relationship.

When identifying Fds between attributes in a relation it is important to distinguish clearly between the values held by an attribute at a given point in time and the set of all possible values that an attributes may hold at different times.

In other words, a functional dependency is a property of a relational schema (its intension) and not a property of a particular instance of the schema (extension).

The reason that we need to identify Fds that hold for all possible values for attributes of a relation is that these represent the types of integrity constraints that we need to identify. Such constraints indicate the limitations on the values that a relation can legitimately assume. In other words, they identify the legal instances which are possible.

Let's identify the functional dependencies that hold using the relation schema STAFFBRANCH

In order to identify the time invariant Fds, we need to clearly understand the semantics of the various attributes in each of the relation schemas in question.

For example, if we know that a staff member's position and the branch at which they are located determines their salary. There is no way of knowing this constraint unless you are familiar with the enterprise, but this is what the requirements analysis phase and the conceptual design phase are all about!

staff# → sname, position, salary, branch#, baddress

branch# → baddress

baddress → branch#

branch#, position → salary

baddress, position → salary

5.3.1Trivial Functional Dependencies

As well as identifying Fds which hold for all possible values of the attributes involved in the fd, we also want to ignore trivial functional dependencies. A functional dependency is trivial if, the consequent is a subset of the determinant. In other words, it is impossible for it not to be satisfied.

Example: Using the relation instances on page 6, the trivial dependencies include:

{ staff#, sname} → sname

{ staff#, sname} → staff#

Although trivial Fds are valid, they offer no additional information about integrity constraints for the relation. As far as normalization is concerned, trivial Fds are ignored.

5.3.2 Inference Rules for Functional Dependencies

We'll denote as F, the set of functional dependencies that are specified on a relational schema R.

Typically, the schema designer specifies the Fds that are semantically obvious; usually however, numerous other Fds hold in all legal relation instances that satisfy the dependencies in F.

These additional Fds that hold are those Fds which can be inferred or deduced from the Fds in F.

The set of all functional dependencies implied by a set of functional dependencies F is called the closure of F and is denoted F+.

The notation: F □ X → Y denotes that the functional dependency X → Y is implied by the set of Fds F.

Formally, F+ ≡ {X → Y | F □ X → Y}

A set of inference rules is required to infer the set of Fds in F+.

For example, if I tell you that Kristi is older than Debi and that Debi is older than Traci, you are able to infer that Kristi is older than Traci. How did you make this inference? Without thinking about it or maybe knowing about it, you utilized a transitivity rule to allow you to make this inference. The set of all Fds that are implied by a given set S of Fds is called the closure of S, written S+. Clearly we need an algorithm that will allow us to compute S+ from S. You know the first attack on this problem appeared in a paper by Armstrong which gives a set of inference rules. The following are the six well-known inference rules that apply to functional dependencies.

IR1: reflexive rule – if $X \supseteq Y$, then $X \rightarrow Y$

IR2: augmentation rule – if $X \rightarrow Y$, then $XZ \rightarrow YZ$

IR3: transitive rule – if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

IR4: projection rule – if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

IR5: additive rule – if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

IR6: pseudo transitive rule – if $X \rightarrow Y$ and $YZ \rightarrow W$, then $XZ \rightarrow W$

The first three of these rules (IR1-IR3) are known as Armstrong's Axioms and constitute a necessary and sufficient set of inference rules for generating the closure of a set of functional dependencies. These rules can be stated in a variety of equivalent ways. Each of these rules can be directly proved from the definition of functional dependency. Moreover the rules are complete, in the sense that, given a set S of Fds, all Fds implied by S can be derived from S using the rules. The other rules are derived from these three rules.

Given R = (A,B,C,D,E,F,G,H, I, J) and

F = {AB → E, AG → J, BE → I, E → G, GI → H}

Does F □ AB → GH?

Proof

1. AB → E, given in F
2. AB → AB, reflexive rule IR1
3. AB → B, projective rule IR4 from step 2
4. AB → BE, additive rule IR5 from steps 1 and 3
5. BE → I, given in F
6. AB → I, transitive rule IR3 from steps 4 and 5
7. E → G, given in F
8. AB → G, transitive rule IR3 from steps 1 and 7
9. AB → GI, additive rule IR5 from steps 6 and 8
10. GI → H, given in F
11. AB → H, transitive rule IR3 from steps 9 and 10
12. AB → GH, additive rule IR5 from steps 8 and 11 – proven

Irreducible sets of dependencies

Let S1 and S2 be two sets of Fds, if every FD implied by S1 is implied by S2- i.e.; if S1+ is a subset of S2+-we say that S2 is a cover for S1+(Cover here means equivalent set). What this means is that if the DBMS enforces the Fds in S2, then it will automatically be enforcing the Fds in S1.

Next…..

Next if S2 is a cover for S1 and S1 is a cover for S2- i.e.; if S1+=S2+ -we say that S1 and S2 are equivalent, clearly, if s1 and S2 are equivalent, then if the DBMS enforces the Fds in S2 it will automatically be enforcing the Fds in S1, And vice versa.

Now we define a set of Fds to be irreducible(Usually called minimal in the literature) if and only if it satisfies the following three properties

1. The right hand side (the dependent) of every Fds in S involves just one attribute (that is, it is singleton set)
2. The left hand side (determinant) of every in S is irreducible in turn-meaning that no attribute can be discarded from the determinant without changing the closure S+(that is, with out converting S into some set not equivalent to S). We will say that such an Fd is **left irreducible**.
3. No Fd in S can be discarded from S without changing the closure S+(That is, without converting s into some set not equivalent to S)

Now we will work out the things in detail.

Relation R {A,B,C,D,E,F} satisfies the following Fds

AB → C

C → A

BC → D

ACD → B

BE → C

CE → FA

CF → VD

D → EF

Find an irreducible equivalent for this set of Fds?

Puzzled! The solution is simple. Let us find the solution for the above.

1. AB → C
2. C → A
3. BC → D
4. ACD → B
5. BE → C

6. CE → A

7. CE → F

8. CF → B

9. CF → D

10. D → E

11. D → F

Now:

• 2 implies 6, so we can drop 6

• 8 implies CF → BC (By augmentation), by which 3 implies CF → D (By Transitivity), so we can drop 10.

• 8 implies ACF → AB (By augmentation), and 11 implies ACD → ACF (By augmentation), and so ACD → AB (By Transitivity), and so ACD → B(By Decomposition), so we can drop 4

No further reductions are possible, and so we are left with the following irreducible set:

AB → C

C → A

BC → D

BE → C

CE → F

CF → B

D → E

D → F

Alternatively:

• 2 implies CD → ACD (By Composition), which with 4 implies CD → BE (By Transitivity), so we can replace 4 CD → B

• 2 implies 6, so we can drop 6(as before)

2 and 10 implies CF → AD (By composition), which implies CF → ADC (By Augmentation), which with (the original) 4 implies CF → B(By Transitivity), So we can drop 8.

No further reductions are possible, and so we are left with following irreducible set:

AB → C

C → A

BC → D

CD → B

BE → C

CE → F

CF → D

D → E

D → F

Observe, therefore, that there are two distinct irreducible equivalence for the original set of Fds.

5.4 Multivalued Dependencies

The multivalued dependency relates to the problem when more than one multivalued attributes exist. Consider the following relation that represents an entity employee that has one mutlivalued attribute proj:

emp (e#, dept, salary, proj)

We have so far considered normalization based on functional dependencies; dependencies that apply only to single-valued facts. For example, **e#→dept** implies only one dept value for each value of e#. Not all information in a database is single-valued, for example, proj in an employee relation may be the list of all projects that the employee is currently working on. Although e# determines the list of all projects that an employee is working on **e#→proj**, is not a functional dependency.

So far we have dealt with multivalued facts about an entity by having a separate relation for that multivalue attribute and then inserting a tuple for each value of that fact. This resulted in composite keys since the multivalued fact must form part of the key. In none of our examples so far have we dealt with an entity having more than one multivalued attribute in one relation. We do so now.

The fourth and fifth normal forms deal with multivalued dependencies. The 4th and 5th normal forms are discussed in the lecture that deals with normalization. We discuss the following example to illustrate the concept of multivalued dependency.

programmer (emp_name, qualifications, languages)

The above relation includes two multivalued attributes of entity programmer; *qualifications* and *languages*. There are no functional dependencies.

The attributes qualifications and languages are assumed independent of each other. If we were to consider qualifications and languages separate entities, we would have two relationships (one between *employees* and *qualifications* and the other between employees and programming languages). Both the above relationships are many-to-many i.e. one programmer could have several qualifications and may know several programming languages. Also one qualification may be obtained by several programmers and one programming language may be known to many programmers.

Suppose a programmer has several qualifications (B.Sc, Dip. Comp. Sc, etc) and is proficient in several programming languages; how should this information be represented? There are several possibilities.

emp name	qualifications	languages
SMITH	B.Sc	FORTRAN
SMITH	B.Sc	COBOL
SMITH	B.Sc	PASCAL
SMITH	Dip.CS	FORTRAN
SMITH	Dip.CS	COBOL
SMITH	Dip.CS	PASCAL

emp name	qualifications	language
SMITH	B.Sc	NULL
SMITH	Dip.CS	NULL
SMITH	NULL	FORTRAN
SMITH	NULL	COBOL
SMITH	NULL	PASCAL

emp name	qualifications	language
SMITH	B.Sc	FORTRAN
SMITH	Dip.CS	COBOL
SMITH	NULL	PASCAL

Figure 5.6

Other variations are possible (we remind the reader that there is no relationship between qualifications and programming languages). All these variations have some disadvantages. If the information is repeated we face the same problems of repeated information and anomalies as we did when second or third normal form conditions are violated. If there is no repetition, there are still some difficulties with search, insertions and deletions. For example, the role of NULL values in the above relations is confusing. Also the candidate key in the above relations is (emp name, qualifications, language) and existential integrity requires that no NULLs be specified. These problems may be overcome by decomposing a relation like the one above(Figure 5.6) as follows:

emp name	qualifications
SMITH	B.Sc
SMITH	Dip.CS

emp name	languages
SMITH	FORTRAN
SMITH	COBOL
SMITH	PASCAL

Figure 5.7

The basis of the above decomposition is the concept of multivalued dependency (MVD). Functional dependency **A→B** relates one value of A to one value of B while multivalued dependency **A→B** defines a relationship in which a set of values of attribute B are determined by a single value of A.

The concept of multivalued dependencies was developed to provide a basis for decomposition of relations like the one above. Therefore if a relation like *enrolment(sno, subject#)* has a relationship between *sno* and *subject#* in which sno uniquely determines the values of *subject#*, the dependence of *subject#* on sno is called a trivial MVD since the relation enrolment cannot be decomposed any further. More formally, a MVD **X→Y** is called *trivial* MVD if either Y is a subset of X or X and Y together form the relation R. The MVD is trivial since it results in no constraints being placed on the relation. Therefore a relation having non-trivial MVDs must have at least three attributes; two of them multivalued. Non-trivial MVDs result in the relation having some constraints on it since all possible combinations of the multivalue attributes are then required to be in the relation.

Let us now define the concept of multivalued dependency.

The multivalued dependency X→Y is said to hold for a relation R(X, Y, Z) if for a given set of value (set of values if X is more than one attribute) for attributes X,

there is a set of (zero or more) associated values for the set of attributes Y and the Y values depend only on X values and have no dependence on the set of attributes Z.

In the example above, if there was some dependence between the attributes *qualifications* and *language*, for example perhaps, the language was related to the qualifications (perhaps the qualification was a training certificate in a particular language), then the relation would not have MVD and could not be decomposed into two relations as above. The theory of multivalued dependencies in very similar to that for functional dependencies. Given D a set of MVDs, we may find $\mathbf{D}^+$, the closure of D using a set of axioms. We do not discuss the axioms here.

5.5 Relational Database

Relational database tables, whether they are derived from ER or UML models, sometimes suffer from some rather serious problems in terms of

Sales

product_name	order_no	cust_name	cust_addr	credit	date	sales_name
vacuum cleaner	1458	Dave Bachmann	Austin	6	1-3-03	Carl Bloch
computer	2730	Qiang Zhu	Plymouth	10	4-15-05	Ted Hanss
refrigerator	2460	Mike Stolarchuck	Ann Arbor	8	9-12-04	Dick Phillips
DVD player	519	Peter Honeyman	Detroit	3	12-5-04	Fred Remley
radio	1986	Charles Antonelli	Chicago	7	5-10-05	R. Metz
CD player	1817	C.V. Ravishankar	Mumbai	8	8-3-02	Paul Basile
vacuum cleaner	1865	Charles Antonelli	Chicago	7	10-1-04	Carl Bloch
vacuum cleaner	1885	Betsy Karmeisool	Detroit	8	4-19-99	Carl Bloch
refrigerator	1943	Dave Bachmann	Austin	6	1-4-04	Dick Phillips
television	2315	Sakti Pramanik	East Lansing	6	3-15-04	Fred Remley

Figure 5.8 Single table database

performance, integrity and maintainability. For example, when the entire database is defined as a single large table, it can result in a large amount of redundant data and lengthy searches for just a small number of target rows. It can also result in long and expensive updates, and deletions in particular can result in the elimination of useful data as an unwanted side effect.

Such a situation is shown in Figure 5.8, where products, salespersons, customers, and orders are all stored in a single table called Sales. In this table, we see that certain product and customer information is stored redundantly, wasting storage space. Certain queries, such as "Which customers ordered vacuum cleaners last month?" would require a search of the entire table. Also, updates such as changing the address of the customer Dave Bachmann would require changing many rows. Finally, deleting an order by a valued customer such as Qiang Zhu (who bought an expensive computer), if that is his only outstanding order, deletes the only copy of his address and credit rating as a side effect. Such information may be difficult (or sometimes impossible) to recover. These problems also occur for situations in which the database has already been set up as a collection of many tables, but some of the tables are still too large.

If we had a method of breaking up such a large table into smaller tables so that these types of problems would be eliminated, the database would be much more efficient and reliable. Classes of relational database schemes or table definitions, called normal forms, are commonly used to accomplish this goal. The creation of a normal form database table is called **normalization**. Normalization is accomplished by analyzing the interdependencies among individual attributes associated with those tables and taking projections (subsets of columns) of larger tables to form smaller ones.

Normalization is a formal process for determining which fields belong in which tables in a relational database. Normalization follows a set of rules worked out at the time relational databases were born. A normalized relational database provides several benefits:

- Elimination of redundant data storage.
- Close modeling of real world entities, processes, and their relationships.
- Structuring of data so that the model is flexible.
- Normalization ensures that you get the benefits relational databases offer. Time spent learning about normalization will begin paying for itself immediately.

Let us first review the basic normal forms, which have been well established in the relational database literature and in practice.

5.6 First Normal Form

Definition. A table is in *first normal form (1NF)* if and only if all columns contain only atomic values, that is, each column can have only one value for each row in the table.

Relational database tables, such as the **Sales** table illustrated in Figure 5.9, have only atomic values for each row and for each column. Such tables are considered to be in first normal form, the most basic level of normalized tables.

To better understand the definition for 1NF, it helps to know the difference between a domain, an attribute, and a column. A *domain* is the set of all possible values for a particular type of attribute, but may be used for more than one attribute. For example, the domain of people's names is the underlying set of all possible names that could be used for either customer-name or salesperson-name in the database table in Figure 5.8. Each column in a relational table represents a single attribute, but in some cases more than one column may refer to different attributes from the same domain. When this occurs, the table is still in 1NF because the values in the table are still atomic. In fact, standard SQL assumes only atomic values and a relational table is by default in 1NF.

5.6.1 Superkeys, Candidate Keys, and Primary Keys

A table in 1NF often suffers from data duplication, update performance, and update integrity problems, as noted above. To understand these issues better, however, we must define the concept of a key in the context of normalized tables. A *superkey* is a set of one or more attributes, which, when taken collectively, allows us to identify uniquely an entity or table. Any subset of the attributes of a superkey that is also a superkey, and not reducible to another superkey, is called a *candidate key*. A *primary* key is selected arbitrarily from the set of candidate keys to be used in an index for that table.

Report

report_no	editor	dept_no	dept_name	dept_addr	author_id	author_name	author_addr
4216	woolf	15	design	argus1	53	mantei	cs-tor
4216	woolf	15	design	argus1	44	bolton	mathrev
4216	woolf	15	design	argus1	71	koenig	mathrev
5789	koenig	27	analysis	argus2	26	fry	folkstone
5789	koenig	27	analysis	argus2	38	umar	prise
5789	koenig	27	analysis	argus2	71	koenig	mathrev

Figure 5.9 Report table

As an example, in Figure 5.9 a composite of all the attributes of the table forms a superkey because duplicate rows are not allowed in the relational model. Thus, a trivial superkey is formed from the composite of all attributes in a table. Assuming that each department address (dept_addr) in this table is single valued, we can conclude that the composite of all attributes except dept_addr is also a superkey. Looking at smaller and smaller composites of attributes and making realistic assumptions about which attributes are single valued, we find that the composite (report_no, author_id) uniquely determines all the other attributes in the table and is therefore a superkey. However, neither report_no nor author_id alone can determine a row uniquely, and the composite of these two attributes cannot be reduced and still be a superkey. Thus, the composite (report_no, author_id) becomes a candidate key. Since it is the only candidate key in this table, it also becomes the primary key.

A table can have more than one candidate key. If, for example, in Figure 5.9, we had an additional column for author_ssn, and the composite of report_no and author_ssn uniquely determine all the other attributes of the table, then both (report_no, author_id) and (report_no, author_ssn) would be candidate keys. The primary key would then be an arbitrary choice between these two candidate keys.

5.7 Second Normal Form

To explain the concept of second normal form (2NF) and higher, we introduce the concept of functional dependence. The property of one or more attributes that uniquely determine the value of one or more other attributes is called *functional dependence*. Given a table (R), a set of attributes (B) is functionally dependent on another set of attributes (A) if, at each instant of time, each A value is associated with only one B value. Such a functional dependence is denoted by A -> B. In the preceding example from Figure 5.9, let us assume we are given the following functional dependencies for the table **report**:

report: report_no -> editor, dept_no

dept_no -> dept_name, dept_addr

author_id -> author_name, author_addr

Definition. A table is in second normal form (2NF) if and only if it is in 1NF and every nonkey attribute is fully dependent on the primary key. An attribute is fully dependent on the primary key if it is on the right side of an FD for which the left side is either the primary key itself or something that can be derived from the primary key using the transitivity of FDs.

An example of a transitive FD in **report** is the following:

report_no -> dept_no

dept_no -> dept_name

Therefore we can derive the FD (report_no -> dept_name), since dept_name is transitively dependent on report_no.

Continuing our example, the composite key in Figure 5.9, (report_no, author_id), is the only candidate key and is therefore the primary key. However, there exists one FD (dept_no -> dept_name, dept_addr) that has no component of the primary key on the left side, and two FDs (report_no -> editor, dept_no and author_id -> author_name, author_addr) that contain one component of the primary key on the left side, but not both components. As such, **report** does not satisfy the condition for 2NF for any of the FDs.

Consider the disadvantages of 1NF in table **report**. Report_no, editor, and dept_no are duplicated for each author of the report. Therefore, if the editor of the report changes, for example, several rows must be updated. This is known as the *update anomaly*, and it represents a potential degradation of performance due to the redundant updating. If a new editor is to be added to the table, it can only be done if the new editor is editing a report: both the report number and editor number must be known to add a row to the table, because you cannot have a primary key with a null value in most relational databases. This is known as the *insert anomaly*. Finally, if a report is withdrawn, all rows associated with that report must be deleted. This has the side effect of deleting the information that associates an author_id with author_name and author_addr. Deletion side effects of this nature are known as *delete anomalies*. They represent a potential loss of integrity, because the only way the data can be restored is to find the data somewhere outside the database and insert it back into the database. All three of these anomalies represent problems to database designers, but the delete anomaly is by far the most serious because you might lose data that cannot be recovered. These disadvantages can be overcome by

transforming the 1NF table into two or more 2NF tables by using the projection operator on the subset of the attributes of the 1NF table. In this example we project **report** over report_no, editor, dept_no, dept_name, and dept_addr to form **report1**; and project **report** over author_id, author_name, and author_addr to form **report2**; and finally project **report** over report_no and author_id to form **report3**. The projection of **report** into three smaller tables has preserved the FDs and the association between report_no and author_no that was important in the original table. Data for the three tables is shown in Figure 5.10. The FDs for these 2NF tables are:

report1: report_no -> editor, dept_no

dept_no -> dept_name, dept_addr

report2: author_id -> author_name, author_addr

report3: report_no, author_id is a candidate key (no FDs)

We now have three tables that satisfy the conditions for 2NF, and we have eliminated the worst problems of 1NF, especially integrity (the delete anomaly). First, editor, dept_no, dept_name, and dept_addr are no longer duplicated for each author of a report. Second, an editor change results in only an update to one row for **report1**. And third, the most important, the deletion of the report does not have the side effect of deleting the author information.

Report 1

report_no	editor	dept_no	dept_name	dept_addr
4216	woolf	15	design	argus 1
5789	koenig	27	analysis	argus 2

Report 2

author_id	author_name	author_addr
53	mantei	cs-tor
44	bolton	mathrev
71	koenig	mathrev
26	fry	folkstone
38	umar	prise
71	koenig	mathrev

Report 3

report_no	author_id
4216	53
4216	44
4216	71
5789	26
5789	38
5789	71

Figure 5.10 2NF tables

Not all performance degradation is eliminated, however; report_no is still duplicated for each author, and deletion of a report requires updates to two tables (**report1** and **report3**) instead of one. However, these are minor problems compared to those in the 1NF table **report**. Note that these three report tables in 2NF could have been generated directly from an ER (or UML) diagram that equivalently modeled this situation with entities Author and Report and a many-to-many relationship between them.

5.8 Third Normal Form

The 2NF tables we established in the previous section represent a significant improvement over 1NF tables. However, they still suffer from

Report 11

report_no	editor	dept_no
4216	woolf	15
5789	koenig	27

Report 12

dept_no	dept_name	dept_addr
15	design	argus 1
27	analysis	argus 2

Report 2

author_id	author_name	author_addr
53	mantei	cs-tor
44	bolton	mathrev
71	koenig	mathrev
26	fry	folkstone
38	umar	prise
71	koenig	mathrev

Report 3

report_no	author_id
4216	53
4216	44
4216	71
5789	26
5789	38
5789	71

Figure 5.11 3NF tables

the same types of anomalies as the 1NF tables although for different reasons associated with transitive dependencies. If a transitive (functional) dependency exists in a table, it means that two separate facts are represented in that table, one fact for each functional dependency involving a different left side. For example, if we delete a report from the database, which involves deleting the appropriate rows from **report1** and **report3** (see Figure 5.10), we have the side effect of deleting the association between dept_no, dept_name, and dept_addr as well. If we could project table **report1** over report_no, editor, and dept_no to form table **report11**, and project **report1** over dept_no, dept_name, and dept_addr to form table **report12**, we could eliminate this problem. Example tables for **report11** and **report12** are shown in Figure 5.11.

Definition. A table is in *third normal form (3NF)* if and only if for every nontrivial functional dependency X->A, where X and A are either simple or composite attributes, one of two conditions must hold. Either attribute X is a superkey, or attribute A is a member of a candidate key. If attribute A is a member of a candidate key, A is called a prime attribute. Note: a trivial FD is of the form YZ->Z. In the preceding example, after projecting **report1** into **report11** and **report12** to eliminate the transitive dependency report_no -> dept_no -> dept_name, dept_addr, we have the following 3NF tables and their functional dependencies (and example data in Figure 5.11):

report11: report_no -> editor, dept_no

report12: dept_no -> dept_name, dept_addr

report2: author_id -> author_name, author_addr

report3: report_no, author_id is a candidate key (no FDs)

5.9 Boyce-Codd Normal Form

3NF, which eliminates most of the anomalies known in databases today, is the most common standard for normalization in commercial databases and CASE tools. The few remaining anomalies can be eliminated by the Boyce-Codd normal form (BCNF). BCNF is considered to be a strong variation of 3NF.

Definition. A table **R** is in *Boyce-Codd normal form (BCNF)* if for every nontrivial FD X->A, X is a superkey.

BCNF is a stronger form of normalization than 3NF because it eliminates the second condition for 3NF, which allowed the right side of the FD to be a prime attribute. Thus, every left side of an FD in a table must be a superkey. Every table that is BCNF is also 3NF, 2NF, and 1NF, by the previous definitions.

The following example shows a 3NF table that is not BCNF. Such tables have delete anomalies similar to those in the lower normal forms.

Assertion 1. For a given team, each employee is directed by only one leader. A team may be directed by more than one leader.

emp_name, team_name -> leader_name

Assertion 2. Each leader directs only one team.

leader_name -> team_name

This table is 3NF with a composite candidate key emp_id, team_id:

team:	emp_name	team_name	leader_name
	Sutton	Hawks	Wei
	Sutton	Condors	Bachmann
	Niven	Hawks	Wei
	Niven	Eagles	Makowski
	Wilson	Eagles	DeSmith

The **team** table has the following delete anomaly: if Sutton drops out of the Condors team, then we have no record of Bachmann leading the Condors team. As shown by Date [1999], this type of anomaly cannot have a lossless decomposition and preserve all FDs. A lossless decomposition requires that when you decompose the table into two smaller tables by projecting the original table over two overlapping subsets of the scheme, the natural join of those subset tables must result in the original table without any extra unwanted rows. The simplest way to avoid the delete anomaly for this kind of situation is to create a separate table for each of the two assertions. These two tables are partially redundant, enough so to avoid the delete anomaly. This decomposition is lossless (trivially) and preserves functional dependencies, but it also degrades update performance due to redundancy, and necessitates additional storage space. The trade-off is often worth it because the delete anomaly is avoided.

5.10 Lossless-Join Decomposition

In this chapter so far we have normalized a number of relations by decomposing them. We decomposed a relation intuitively. We need a better basis for deciding decompositions since intuition may not always be correct. We illustrate how a careless decomposition may lead to problems including loss of information.

Consider the following relation

enrol (sno, cno, date-enrolled, room-No., instructor)

Suppose we decompose the above relation into two relations enrol1 and enrol2 as follows

enrol1 (sno, cno, date-enrolled)

enrol2 (date-enrolled, room-No., instructor)

There are problems with this decomposition but we wish to focus on one aspect at the moment. Let an instance of the relation enrol be

sno	cno	date-enrolled	room-No.	instructor
830057	CP302	1FEB1984	MP006	Gupta
830057	CP303	1FEB1984	MP006	Jones
820159	CP302	10JAN1984	MP006	Gupta
825678	CP304	1FEB1984	CE122	Wilson
826789	CP305	15JAN1984	EA123	Smith

and let the decomposed relations enrol1 and enrol2 be:

sno	cno	date-enrolled
830057	CP302	1FEB1984
830057	CP303	1FEB1984
820159	CP302	10JAN1984
825678	CP304	1FEB1984
826789	CP305	15JAN1984

date-enrolled	room-No.	instructor
1FEB1984	MP006	Gupta
1FEB1984	MP006	Jones
10JAN1984	MP006	Gupta
1FEB1984	CE122	Wilson
15JAN1984	EA123	Smith

All the information that was in the relation enrol appears to be still available in enrol1 and enrol2 but this is not so. Suppose, we wanted to retrieve the student numbers of all students taking a course from Wilson, we would need to join enrol1 and enrol2. The join would have 11 tuples as follows:

sno	cno	date-enrolled	room-No.	instructor
830057	CP302	1FEB1984	MP006	Gupta
830057	CP302	1FEB1984	MP006	Jones
830057	CP303	1FEB1984	MP006	Gupta
830057	CP303	1FEB1984	MP006	Jones
830057	CP302	1FEB1984	CE122	Wilson
830057	CP303	1FEB1984	CE122	Wilson

(add further tuples ...)

The join contains a number of spurious tuples that were not in the original relation Enrol. Because of these additional tuples, we have lost the information about which students take courses from WILSON. (Yes, we have more tuples but less information because we are unable to say with certainty who is taking courses from WILSON). Such decompositions are called lossy decompositions. A nonloss or lossless decomposition is that which guarantees that the join will result in exactly the same relation as was decomposed. One might think that there might be other ways of recovering the original relation from the decomposed relations but, sadly, no other operators can recover the original relation if the join does not (why?).

We need to analyse why some decompositions are lossy. The common attribute in above decompositions was Date-enrolled. The common attribute is the glue that gives us the ability to find the relationships between different relations by joining the relations together. If the common attribute is not unique, the relationship information is not preserved. If each tuple had a unique value of Date-enrolled, the problem of losing information would not have existed. The problem arises because several enrolments may take place on the same date.

A decomposition of a relation R into relations $\mathbf{R_1, R_2,\ldots..,R_n}$ is called a lossless-join decomposition (with respect to FDs F) if the relation R is always the natural join of the relations $\mathbf{R_1, R_2,\ldots..,R_n}$. It should be noted that natural join is the only way to recover the relation from the decomposed relations. There is no other set of operators that can recover the relation if the join cannot. Furthermore, it should be noted when the decomposed relations $\mathbf{R_1, R_2,\ldots..,R_n}$ are obtained by projecting on the relation R, for example $\mathbf{R_1}$ by projection $\mathbf{\Pi_{1(R)}}$, the relation $\mathbf{R_1}$ may not always be precisely equal to the projection since the relation $\mathbf{R_1}$ might have additional tuples called the dangling tuples. Explain ...

It is not difficult to test whether a given decomposition is lossless-join given a set of functional dependencies F. We consider the simple case of a relation R being decomposed into $\mathbf{R_1}$ and $\mathbf{R_2}$. If the decomposition is lossless-join, then one of the following two conditions must hold

$\mathbf{R_1 \cap R2 \rightarrow R_{1\,-\,R2}}$

$\mathbf{R_1 \cap R2 \rightarrow R_{2\,-\,R1}}$

5.11 Dependency Preservation Decomposition

It is clear that decomposition must be lossless so that we do not lose any information from the relation that is decomposed. Dependency preservation is another important requirement since a dependency is a constraint on the database and if **x→y** holds than we know that the two (sets) attributes are closely related and it would be useful if both attributes appeared in the same relation so that the dependency can be checked easily.

Let us consider a relation R(A, B, C, D) that has the dependencies F that include the following:

A→ B

A →C

etc

If we decompose the above relation into R1(A, B) and R2(B, C, D) the dependency **A →C** cannot be checked (or preserved) by looking at only one relation. It is desirable that decompositions be such that each dependency in F may be checked by looking at only one relation and that no joins need be computed for checking dependencies. In some cases, it may not be possible to preserve each and every dependency in F but as long as the dependencies that are preserved are equivalent to F, it should be sufficient.

Let F be the dependencies on a relation R which is decomposed in relations $\mathbf{R_1, R_2,\ldots..,R_n}$.

We can partition the dependencies given by F such that $\mathbf{F_1,F_2,\ldots..F_N}$. $\mathbf{F_N}$ are dependencies that only involve attributes from relations $\mathbf{R_1, R_2,\ldots..,R_n}$ respectively. If the union of dependencies $\mathbf{F_i}$ imply all the dependencies in F, then we say that the decomposition has preserved dependencies, otherwise not.

If the decomposition does not preserve the dependencies F, then the decomposed relations may contain relations that do not satisfy F or the updates to the decomposed

relations may require a join to check that the constraints implied by the dependencies still hold.

Consider the following relation

sub(sno, instructor, office)

We may wish to decompose the above relation to remove the transitive dependency of office on sno. A possible decomposition is

S1(sno, instructor)

S2(sno, office)

The relations are now in 3NF but the dependency ***instructor→office*** cannot be verified by looking at one relation; a join of S1 and S2 is needed. In the above decomposition, it is quite possible to have more than one office number for one instructor although the functional dependency ***instructor→office*** does not allow it.

5.12 Summary

1. While designing a relational schema semantics of attributes, reducing the redundant values in a tuple, reducing null valuesin tuples and avoiding generation of spurious tuples are some of the issues that need to be taken care of.
2. Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations.
3. Redundancy arises when a relational schema forces an association between attributes that is not natural. Functional dependencies can 'be used to identify such situations and suggest refinements to the schema.
4. A functional dependency is a property of the semantics of the attributes in a relation. The semantics indicate how attributes relate to one another, and specify the functional dependencies between attributes.
5. A table is in *first normal form (1NF)* if and only if all columns contain only atomic values, that is, each column can have only one value for each row in the table.
6. A *superkey* is a set of one or more attributes, which, when taken collectively, allows us to identify uniquely an entity or table.

7. Any subset of the attributes of a superkey that is also a superkey, and not reducible to another superkey, is called a *candidate key*.
8. A *primary* key is selected arbitrarily from the set of candidate keys to be used in an index for that table.
9. A table **R** is in *Boyce-Codd normal form (BCNF)* if for every nontrivial FD X->A, X is a superkey.

5.13 Key Words

Database Design, Modification Anomalies, Decomposition, Functional Dependency, Normalisation, First Normal Form, Second Normal Form, Third Normal Form, BCNF, Lossless Join, Dependency Preservation, Super key, Candidate Key, Primary Key

5.14 Self Assessment Questions

1. What are the various guidelines that need to be taken care of while designing a relational schema?
2. Describe update, insert and delete anomalies with the help of examples.
3. Define functional dependencies?
4. Explain the inference rules?
5. Explain the concept of multi valued dependency?
6. What does the term unnormalized relation refer to? How did the normal forms develop historically?
7. Write down all the rules for normalization and explain with example.
8. Define first, second, and third normal forms when only primary keys are considered. How do the general definitions of 2NF and 3NF, which consider all keys of a relation, differ from those that consider only primary keys?

5.15 References/Suggested Readings

1. http://www.microsoft-accesssolutions.co.uk
2. Date, C, J, Introduction to Database Systems, 7th edition
3. Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld.

MCA 202/MS 11

Author: Abhishek Taneja **Vetter: Prof. Dharminder Kumar**
Lesson: Query Processing **Lesson No. : 06**

Structure

6.0 Objectives

At the end of this chapter the reader will be able to:

- Describe fundamentals of Query Processing
- Describe Query Optimization
- Describe Heuristics in Query Optimization
- Describe various algorithms for query processing

6.1 Inroduction

In this chapter we would like to discuss with you in detail about the query processing in DBMS. In this lesson you will find many repetitions from the previous chapters. That is included in this lesson purposefully in order to maintain the continuity and meaningfulness of the topic which we are going to deal with. So let's start the lecture with a bang.

In most database systems, queries are posed in a non-procedural language like SQL and as we have noted earlier such queries do not involve any reference to access paths or the order of evaluation of operations. The query processing of such queries by a DBMS usually involves the following four phases:

1. Parsing
2. Optimization
3. Code Generation
4. Execution

The parser basically checks the query for correct syntax and translates it into a conventional parse-tree (often called a query-tree) or some other internal representation.

If the parser returns with no errors, and the query uses some user-defined views, it is necessary to expand the query by making appropriate substitutions for the views. It is then necessary to check the query for semantic correctness by consulting the system catalogues and check for semantic errors and type compatibility in both expressions and predicate comparisons. The optimizer is then invoked with the internal representation of the query as input so that a query plan or execution plan may be devised for retrieving the information that is required. The optimizer carries out a number of operations. It relates the symbolic names in the query to data base objects and checks their existence and checks if the user is authorized to perform the operations that the query specifies.

In formulating the plans, the query optimizer obtains relevant information from the metadata that the system maintains and attempts to model the estimated costs of performing many alternative query plans and then selects the best amongst them. The

metadata or system catalog consists of descriptions of all the databases that a DBMS maintains. Often, the query optimizer would at least retrieve the following information:

1. Cardinality of each relation of interest.
2. The number of pages in each relation of interest.
3. The number of distinct keys in each index of interest.
4. The number of pages in each index of interest.

The above information and perhaps other information will be used by the optimizer in modeling the cost estimation for each alternative query plan.

Considerable other information is normally available in the system catalog:

1. Name of each relation and all its attributes and their domains.
2. Information about the primary key and foreign keys of each relation.
3. Descriptions of views.
4. Descriptions of storage structures.
5. Other information including information about ownership and security issues.

Often this information is updated only periodically and not at every update/insert/delete. Also, the system catalog is often stored as a relational database itself making it easy to query the catalog if a user is authorized to do so.

Information in the catalog is very important of course since query processing makes use of this information extensively. Therefore more comprehensive and more accurate information a database maintains the better optimization it can carry out but maintaining more comprehensive and more accurate information also introduces additional overheads and a good balance therefore must be found.

The catalog information is also used by the optimizer in access path selection. These statistics are often updated only periodically and are therefore not always accurate.

An important part of the optimizer is the component that consults the metadata stored in the database to obtain statistics about the referenced relations and the access paths available on them. These are used to determine the most efficient order of the relational operations and the most efficient access paths. The order of operations and the access

paths are selected from a number of alternate possibilities that normally exist so that the cost of query processing is minimized. More details of query optimization are presented in the next section.

If the optimizer finds no errors and outputs an execution plan, the code generator is then invoked. The execution plan is used by the code generator to generate the machine language code and any associated data structures. This code may now be stored if the code is likely to be executed more than once. To execute the code, the machine transfers control to the code which is then executed.

6.2 Query Optimization

Before query optimization is carried out, one would of course need to decide what needs to be optimized. The goal of achieving efficiency itself may be different in different situations. For example, one may wish to minimize the processing time but in many situations one would wish to minimize the response time. In other situations, one may wish to minimize the I/O, network time, memory used or some sort of combination of these e.g. total resources used. Generally, a query processing algorithm A will be considered more efficient than an algorithm B if the measure of cost being minimized for processing the same query given the same resources using A is generally less than that for B.

To illustrate the desirability of optimization, we now present an example of a simple query that may be processed in several different ways. The following query retrieves subject names and instructor names of all current subjects in Computer Science that John Smith is enrolled in.

SELECT *subject.name, instructor* FROM *student, enrolment, subject* WHERE *student.student_id = enrolment.student_id* AND *subject.subject_id = nrolment.subject_id* AND *subject.department = `Computer Science'* AND *student.name = `John Smith'*

To process the above query, two joins and two restrictions need to be performed. There are a number of different ways these may be performed including the following:

1. Join the relations *student* and *enrolment*, join the result with *subject* and then do the restrictions.

2. Join the relations *student* and *enrolment*, do the restrictions, join the result with *subject*

3. Do the restrictions, join the relations *student* and *enrolment*, join the result with *subject*

4. Join the relations *enrolment* and *subject*, join the result with *student* and then do the restrictions.

Here we are talking about the cost estimates. Before we attempt to compare the costs of the above four alternatives, it is necessary to understand that estimating the cost of a plan is often non- trivial. Since normally a database is disk-resident, often the cost of reading and writing to disk dominates the cost of processing a query. We would therefore estimate the cost of processing a query in terms of disk accesses or block accesses. Estimating the number of block accesses to process even a simple query is not necessarily straight forward since it would depend on how the data is stored and which, if any, indexes are available. In some database systems, relations are stored in packed form, that is, each block only has tuples of the same relation while other systems may store tuples from several relations in each block making it much more expensive to scan all of a relation.

Let us now compare the costs of the above four options. Since exact cost computations are difficult, we will use simple estimates of the cost. We consider a situation where the enrolment database consists of 10,000 tuples in the relation *student*, 50,000 in *enrolment*, and 1,000 in the relation *subject*. For simplicity, let us assume that the relations student and subject have tuples of similar size of around 100 bytes each and therefore and we can accommodate 10 tuples per block if the block is assumed to be 1 Kbytes in size. For the relation enrolment, we assume a tuple size of 40 bytes and thus we use a figure of 25 tuples/block. In addition, let John Smith be enrolled in 10 subjects and let there be 20 subjects offered by Computer Science. We can now estimate the costs of the four plans listed above.

The cost of query plan (1) above may now be computed. Let the join be computed by reading a block of the first relation followed by a scan of the second relation to identify matching tuples (this method is called nested-scan and is not particularly efficient. We will discuss the issue of efficiency of algebraic operators in a later section). This is then followed by the reading of the second block of the first relation followed by a scan of the second relation and so on. The cost of R |X| S may therefore be estimated as the number of blocks in R times the number of blocks in S. Since the number of blocks in student is 1000 and in enrolment 2,000, the total number of blocks read in computing the join of student and enrolment is **1000X 2000=2,000,000** block accesses. The result of the join is 50,000 tuples since each tuple from enrolment matches with a tuple from student. The joined tuples will be of size approximately 140 bytes since each tuple in the join is a tuple from student joined with another from enrolment. Given the tuple size of 140 bytes, we can only fit 7 tuples in a block and therefore we need about 7,000 blocks to store all 50,000 joined tuples. The cost of computing the join of this result with subject is **7000 X 100= 700,00** block accesses. Therefore the total cost of plan (1) is approximately 2,700,000 block accesses.

To estimate the cost of plan (2), we know the cost of computing the join of student and enrolment has been estimated above as 2,000,000 block accesses. The result is 7000 blocks in size. Now the result of applying the restrictions to the result of the join reduces this result to about 5-10 tuples i.e. about 1-2 blocks. The cost of this restriction is about 7000 disk accesses. Also the result of applying the restriction to the relation subject reduces that relation to 20 tuples (2 blocks). The cost of this restriction is about 100 block accesses. The join now only requires about 4 block accesses. The total cost therefore is approximately 2,004,604.

To estimate the cost of plan (3), we need to estimate the size of the results of restrictions and their cost. The cost of the restrictions is reading the relations student and subject and writing the results. The reading costs are 1,100 block accesses. The writing costs are very small since the size of the results is 1 tuple for student and 20 tuples for subject. The cost of computing the join of student and enrolment primarily involves the cost of reading enrolment. This is 2,000 block accesses. The result is quite small in size and therefore the

cost of writing the result back is small. The total cost of plan (3) is therefore 3,100 block accesses.

Similar estimates may be obtained for processing plan (4). We will not estimate this cost, since the above estimates are sufficient to illustrate that brute force method of query processing is unlikely to be efficient. The cost can be significantly reduced if the query plan is optimized. The issue of optimization is of course much more complex than estimating the costs like we have done above since in the above estimation we did not consider the various alternative access paths that might be available to the system to access each relation.

The above cost estimates assumed that the secondary storage access costs dominate the query processing costs. This is often a reasonable assumption although the cost of communication is often quite important if we are dealing with a distributed system. The cost of storage can be important in large databases since some queries may require large intermediate results. The cost of CPU of course is always important and it is not uncommon for database applications to be CPU bound than I/O bound as is normally assumed. In the present chapter we assume a centralized system where the cost of secondary storage access is assumed to dominate other costs although we recognize that this is not always true. For example, system R uses

cost = page fetches + w cpu utilization

When a query is specified to a DBMS, it must choose the best way to process it given the information it has about the database. The optimization part of query processing generally involves the following operations.

1. A suitable internal representation
2. Logical transformation of the query
3. Access path selection of the alternatives
4. Estimate costs and select best

We will discuss the above steps in detail.

6.2.1 Internal Representation

As noted earlier, a query posed in a query language like SQL must first be translated to a internal representation suitable for machine representation. Any internal query representation must be sufficiently powerful to represent all queries in the query language (e.g. SQL). The internal representation could be relational algebra or relational calculus since these languages are powerful enough (they have been shown to be relationally complete by E.F. Codd) although it will be necessary to modify them from what was discussed in an earlier chapter so that features like Group By and aggregations may be represented. A representation like relational algebra is procedural and therefore once the query is represented in that representation, a sequence of operations is clearly indicated. Other representations are possible. These include object graph, operator graph (or parse tree) and tableau. Further information about other representations is available in Jarke and Koch (1984) although some sort of tree representation appears to be most commonly used (why?). Our discussions will assume that a query tree representation is being used. In such a representation, the leaf nodes of the query tree are the base relations and the nodes correspond to relational operations.

6.2.2 Logical Transformations

At the beginning of this chapter we showed that the same query may be formulated in a number of different ways that are semantically equivalent. It is clearly desirable that all such queries be transformed into the same query representation. To do this, we need to translate each query to some canonical form and then simplify.

This involves transformations of the query and selection of an optimal sequence of operations. The transformations that we discuss in this section do not consider the physical representation of the database and are designed to improve the efficiency of query processing whatever access methods might be available. An example of such transformation has already been discussed in the examples given. If a query involves one or more joins and a restriction, it is always going to be more efficient to carry out the restriction first since that will reduce the size of one of the relations (assuming that the

restriction applies to only one relation) and therefore the cost of the join, often quite significantly.

Heuristic Optimization -- In the heuristic approach, the sequence of operations in a query is reorganized so that the query execution time improves.

Deterministic Optimization -- In the deterministic approach, cost of all possible forms of a query are evaluated and the best one is selected.

Common Subexpression -- In this technique, common subexpressions in the query, if any, are recognised so as to avoid executing the same sequence of operations more than once.

6.3 Heuristic in Query optimization

Heuristic optimization often includes making transformations to the query tree by moving operators up and down the tree so that the transformed tree is equivalent to the tree before the transformations. Before we discuss these heuristics, it is necessary to discuss the following rules governing the manipulation of relational algebraic expressions:

1. Joins and Products are commutative. e.g.

R X S = S X R

R |X| S = S |X| R

where |X| may be a join or a natural join. The order of attributes in the two products or joins may not be quite the same but the ordering of attributes is not considered significant in the relational model since the attributes are referred to by their name not by their position in the list of attributes.

2. Restriction is commutative. e.g.

$$\sigma_{p}(\sigma_{q}(R)) = \sigma_{q}(\sigma_{p}(R))$$

3. Joins and Products are associative. e.g.

(R X S) X T = R X (S X T)

(R |X| S) |X| T = R |X| (S |X| T)

The associativity of the above operations guarantees that we will obtain the same results whatever be the ordering of computations of the operations product and join. Union and intersection are also associative.

4. Cascade of Projections. e.g.

$$\pi_A(\pi_B(R)) = \pi_A(R)$$

where the attributes A is a subset of the attributes B. The above expression formalises the obvious that there is no need to take the projection with attributes B if there is going to be another projection which is a subset of B that follows it.

5. Cascade of restrictions. e.g.

$$\sigma_p(\sigma_q(R)) = \sigma_{p \wedge q}(R)$$

The above expression also formalises the obvious that if there are two restrictions, one after the other, then there is no need to carry out the restrictions one at a time (since each will require processing a relation) and instead both restrictions could be combined.

6. Commuting restrictions and Projections. e.g.

$$\sigma_p(\pi_A(R)) = \pi_A \sigma_p(R)$$

or

$$\pi_A \sigma_p(R) = \sigma_p(\pi_A(R))$$

There is no difficulty in computing restriction with a projection since we are then doing the restriction before the projection. However if we wish to commute the projection and the restriction, that is possible only if the restriction used no attributes other than those that are in the projection.

7. **Commuting restrictions with Cartesian Product. In some cases, it is possible to apply commutative law to restrictions and a product. For example,**

$$\sigma_p(R X S) = \sigma_p(R) X S$$

or

$$\sigma_{p \wedge q}(R X S) = \sigma_p(R) X \sigma_q(S)$$

In the above expressions we have assumed that the predicate p has only attributes from R and the predicate q has attributes from S only.

8. **Commuting restriction with a Union.**
9. **Commuting restriction with a Set Difference.**
10. **Commuting Projection with a Cartesian Product or a Join -- we assume that the projection includes the join predicate attributes.**
11. **Commuting Projection with a UNION**.

We now use the above rules to transform the query tree to minimize the query cost. Since the cost is assumed to be closely related to the size of the relations on which the operation is being carried out, one of the primary aims of the transformations that we discuss is to reduce the size of intermediate relations.

The basic transformations include the following:

(a) *Moving restrictions down the tree as far as possible*. The idea is to carry out restrictions as early as possible. If the query involves joins as well as restrictions, moving the restrictions down is likely to lead to substantial savings since the relations that are joined after restrictions are likely to be smaller (in some cases much smaller) than before restrictions. This is clearly shown by the example that we used earlier in this chapter to show that some query plans can be much more expensive than others. The query plans that cost the least were those in which the restriction was carried out first. There are of course situations where a restriction does not reduce the relation significantly, for example, a restriction selecting only women from a large relation of customers or clients.

(b) *Projections are executed as early as possible*. In real-life databases, a relation may have one hundred or more attributes and therefore the size of each tuple is relatively large. Some relations can even have attributes that are images making each tuple in such relations very large. In such situations, if a projection is executed early and it leads to elimination of many attributes so that the resulting relation has tuples of much smaller size, the amount of data that needs to be read in from the disk for the operations that follow could be reduced substantially leading to cost savings. It should be noted that only attributes that we need to retain from the relations are those that are either needed for the result or those that are to be used in one of the operations that is to be carried out on the relation.

(c) *Optimal Ordering of the Joins*. We have noted earlier that the join operator is associative and therefore when a query involves more than one join, it is necessary to find an efficient ordering for carrying out the joins. An ordering is likely to be efficient if we carry out those joins first that are likely to lead to small results rather than carrying out those joins that are likely to lead to large results.

(d) *Cascading restrictions and Projections*. Sometimes it is convenient to carry out more than one operations together. When restrictions and projections have the same operand, the operations may be carried out together thereby saving the cost of scanning the relations more than once.

(e) *Projections of projections are merged into one projection*. Clearly, if more than one projection is to be carried out on the same operand relation, the projections should be merged and this could lead to substantial savings since no intermediate results need to be written on the disk and read from the disk.

(f) *Combining certain restrictions and Cartesian Product to form a Join.* A query may involve a cartesian product followed by a restriction rather than specifying a join. The optimizer should recognise this and execute a join which is usually much cheaper to perform.

(g) *Sorting is deferred as much as possible*. Sorting is normally a nlogn operation and by deferring sorting, we may need to sort a relation that is much smaller than it would have been if the sorting was carried out earlier.

(h) A set of operations is reorganised using commutativity and distribution if a reorganised form is more efficient.

6.4 Basic Algorithms for Executing Query Operations

The efficiency of query processing in a relational database system depends on the efficiency of the relational operators. Even the simplest operations can often be executed in several different ways and the costs of the different ways could well be quite different. Although the join is a frequently used and the most costly operator and therefore worthy of detailed study, we also discuss other operators to show that careful thought is needed in efficiently carrying out the simpler operators as well.

Selection

Let us consider the following simple query:

SELECT *A*

FROM *R*

WHERE *p*

The above query may involve any of a number of types of predicates. The following list is presented by Selinger et al: [could have a query with specifying WHERE condition in different ways]

1**. attribute = value**

2. attribute1 = attribute2

3. attribute > value

4. attribute between value1 and value2

5. attribute IN (list of values)

6. attribute IN subquery

7. predicate expression OR predicate expression

8. predicate expression AND predicate expression

Even in the simple case of equality, two or three different approaches may be possible depending on how the relation has been stored. Traversing a file to find the information of interest is often called a *file scan* even if the whole file is not being scanned. For example, if the predicate involves an equality condition on a single attribute and there is an index on that attribute, it is most efficient to search that index and find the tuple where the attribute value is equal to the value given. That should be very efficient since it will only require accessing the index and then one block to access the tuple. Of course, it is possible that there is no index on the attribute of interest or the condition in the WHERE clause is not quite as simple as an equality condition on a single attribute. For example, the condition might be an inequality or specify a range. The index may still be useful if one exists but the usefulness would depend on the condition that is posed in the WHERE clause. In some situations it will be necessary to scan the whole relation R to find the tuples that satisfy the given condition. This may not be so expensive if the relation is not so large and the tuples are stored in packed form but could be very expensive if the relation is large and the tuples are stored such that each block has tuples from several different relations. Another possibility is of course that the relation R is stored as a hash file using the attribute of interest and then again one would be able to hash on the value specified and find the record very efficiently.

As noted above, often the condition may be a conjunction or disjunction of several different conditions i.e. it may be like $\mathbf{P_1}$ **AND** $\mathbf{P_2}$ or $\mathbf{P_1}$ **OR** $\mathbf{P_2}$. Sometime such conjunctive queries can be efficiently processed if there is a composite index based on the attributes that are involved in the two conditions but this is an exception rather than a rule. Often however, it is necessary to assess which one of the two or more conditions can be processed efficiently. Perhaps one of the conditions can be processed using an index. As a first step then, those tuples that satisfy the condition that involves the most efficient search (or perhaps that which retrieves the smallest number of tuples) are retrived and the remaining conditions are then tested on the tuples that are retrieved. Processing disjunctive queries of course requires somewhat different techniques since in this case we are looking at a union of all tuples that satisfy any one of the conditions and therefore

each condition will need to be processed separately. It is therefore going to be of little concern which of the conditions is satisfied first since all must be satisfied independently of the other. Of course, if any one of the conditions requires a scan of the whole relation then we can test all the conditions during the scan and retrieve all tuples that satisfy any one or more conditions.

Projection

A projection would of course require a scan of the whole relation but if the projection includes a candidate key of the relation then no duplicate removal is necessary since each tuple in the projection is then guaranteed to be unique. Of course, more often the projection would not include any candidate key and may then have duplicates. Although many database systems do not remove duplicates unless the user specifies so, duplicates may be removed by sorting the projected relation and then identifying the duplicates and eliminating them. It is also possible to use hashing which may be desirable if the relations are particularly large since hashing would hash identical tuples to the same bucket and would therefore only require sorting the relations in each bucket to find the duplicates if any.

Often of course one needs to compute a restriction and a join together. It is then often appropriate to compute the restriction first by using the best access paths available (e.g. an index).

Join

We assume that we wish to carry out an equi-join of two relations R and S that are to be joined on attributes a in R and b in S. Let the cardinality of R and S be m and n respectively. We do not count join output costs since these are identical for all methods. We assume $|\mathbf{R}| <= |\mathbf{S}|$. We further assume that all restrictions and projections of R and S have already been carried out and neither R nor S is ordered or indexed unless explicitly noted.

Because of the importance of the join operator in relational database systems and the fact that the join operator is considerably more expensive than operators like selection and projection, a number of algorithms have been suggested for processing the join. The more commonly used algorithms are:

1. The Nested Scan Method

2. The Sort-Merge algorithm

3. Hashing algorithm (hashing no good if not equi-join?)

4. Variants of hashing

5. Semi-joins

6. Filters

7. Links

8. Indexes

9. More recently, the concept of join indices has been proposed by Valduriez (1987). Hashing methods are not good when the join is not an equi-join.

6.4.1 Nested Iteration

Before discussing the methods listed above, we briefly discuss the naive nested iteration method that accesses every pair of tuples and concatenates them if the equi-join condition (or for that matter, any other condition) is satisfied. The cost of the naive algorithm is *O(mn)* assuming that *R* and *S* both are not ordered. The cost obviously can be large when *m* and *n* are large.

We will assume that the relation R is the *outer relation*, that is, R is the relation whose tuples are retrieved first. The relation S is then the *inner relation* since in the nested iteration loop, tuples of S will only be retrieved when a tuple of R has been read. A predicate which related the join attributes is called the join predicate. The algorithm may be written as:

for i = 1 to m

do access *ith* tuple of *R*;

for j = 1 to *n* do

access *jth* tuple of *S*;

compare *ith* tuple of *R* and the *jth* tuple of *S*;

if equi-join condition is satisfied then concatenate and save;

end

end.

This method basically scans the outer relation *(R)* first and retrieves the first tuple. The entire inner relation *S* is then scanned and all the tuples of S that satisfy the join predicate with the first tuple of *R* are combined with that tuple of *R* and output as result. The process then continues with the next tuple of *R* until *R* is exhausted.

This has cost *(m + mn)* which is order*(mn)*. If the memory buffers can store two blocks, one from R and one from S, the cost will go down by a factor rs where r and s are the number of tuples per block in R and S respectively. The technique is sometimes called the nested block method. Some cost saving is achieved by reading the smaller relation in the outer block since this reduces *(m + mn)*. The cost of the method would of course be much higher if the relations are not stored in a packed form since then we might need to retrieve many more tuples.

Efficiency of the nested iteration (or nested block iteration) would improve significantly if an index was available on one of the join attributes. If the average number of blocks of relation S accessed for each tuple of *R* was *c* then the cost of the join would be *(m + mc)* where **c< n**.

6.4.2 Using Indexes

The nested iteration method can be made more efficient if indexes are available on both join columns in the relations *R* and *S*.

Assume that we have available indexes on both join columns *a* and *b* in the relations *R* and *S* respectively. We may now scan both the indexes to determine whether a pair of tuples has the same value of the join attribute. If the value is the same, the tuple from *R* is selected and then all the tuples from *S* are selected that have the same join attribute value. This is done by scanning the index on the join attribute in *S*. The index on the join attribute in *R* is now scanned to check if there are more than the one tuple with the same value of the attribute. All the tuples of R that have the same join attribute value are then selected and combined with the tuples of S that have already been selected. The process then continues with the next value for which tuples are available in *R* and *S*.

Clearly this method requires substantial storage so that we may store all the attributes from *R* and *S* that have the same join attribute value. The cost of the join when the indexes are used may be estimated as follows. Let the cost of reading the indexes be $\mathbf{aN_1}$ and B$\mathbf{N_2}$, then the total cost is

. $\mathbf{aN_1 + BN_2 + N_1 + N_2}$

Cost savings by using indexes can be large enough to justify building an index when a join needs to be computed.

6.4.3 The Sort Merge Method

The nested scan technique is simple but involves matching each block of *R* with every block of *S*. This can be avoided if both relations were ordered on the join attribute. The sort-merge algorithm was introduced by Blasgen and Eswaran in 1977. It is a classical technique that has been the choice for joining relations that have no index on either of the two attributes.

This method involves sorting the relations R and S on the join attributes (if not already sorted), storing them as temporary lists and then scanning them block by block and merging those tuples that satisfy the join condition. The advantage of this scheme is that all of the inner relation (in the nested iteration) does not need to be read in for each tuple of the outer relation. This saving can be substantial if the outer relation is large.

Let the cost of sorting R and S be $\mathbf{C_r}$ and **Cs** and let the cost of reading the two relations in main memory be $\mathbf{N_r}$ and $\mathbf{N_s}$ respectively. The total cost of the join is then

$\mathbf{C_r + Cs + N_r + N_s}$

If one or both the relations are already sorted on the join attribute then the cost of the join reduces.

The algorithm can be improved if we use Multiway Merge-Sort

The cost of sorting is $\mathbf{n \log_n}$.

6.4.4 Simple Hash Join Method

This method involves building a hash table of the smaller relation *R* by hashing each tuple on its hash attribute. Since we have assumed that the relation *R* is too large to fit in the main memory, the hash table would in general not fit into the main memory. The hash table therefore must be built in stages. A number of addresses of the hash table are first selected such that the tuples hashed to those addresses can be stored in the main memory. The tuples of *R* that do not hash to these addresses are written back to the disk. Let these tuples be relation **R'**. Now the algorithm works as follows:

(a) Scan relation *R* and hash each tuple on its join attribute. If the hashed value is equal to one of the addresses that are in the main memory, store the tuple in the hash table. Otherwise write the tuple back to disk in a new relation **R'**.

(b) Scan the relation *S* and hash each tuple of *S* on its join attribute. One of the following three conditions must hold:

1. The hashed value is equal to one of the selected values, and one or more tuple of *R* with same attribute value exists. We combine the tuples of R that match with the tuple of *S* and output as the next tuples in the join.

2. The hashed value is equal to one of the selected values, but there is no tuple in *R* with same join attribute value. These tuple of S are rejected.

3. The hashed value is not equal to one of the selected values. These tuples are written back to disk as a new relation *S'*.

The above step continues till *S* is finished.

(c) Repeat steps (a) and (b) until either relation **R'** or **S'** or both are exhausted.

6.4.5 Grace Hash-Join Method

This method is a modification of the Simple Hash Join method in that the partitioning of *R* is completed before *S* is scanned and partitioning of S is completed before the joining phase. The method consists of the following three phases:

1. Partition *R* - Since R is assumed to be too large to fit in the main memory, a hash table for it cannot be built in the main memory. The first phase of the algorithm involves partitioning the relation into *n* buckets, each bucket corresponding to a hash table entry. The number of buckets *n* is chosen to be large enough so that each bucket will comfortably fit in the main memory.

2. Partition *S* - The second phase of the algorithm involves partitioning the relation S into the same number *(n)* of buckets, each bucket corresponding to a hash table entry. The same hashing function as for R is used.

3. Compute the Join - A bucket of *R* is read in and the corresponding bucket of *S* is read in. Matching tuples from the two buckets are combined and output as part of the join.

6.4.6 Hybrid Hash Join Method

The hybrid hash join algorithm is a modification of the Grace hash join method.

Aggregation

Aggregation is often found in queries given the frequency of requirements of finding an average, the maximum or how many times something happens. The functions supported in SQL are average, minimum, maximum, count, and sum. Aggregation can itself be of different types including aggregation that only requires one relation, for example finding the maximum mark in a subject, or it may involve a relation but require something like finding the number of students in each class. The latter aggregation would obviously require some grouping of the tuples in the relation before aggregation can be applied.

6.5 Summary

1. The query processing by a DBMS usually involves the four phases Parsing, Optimization, Code Generation, and Execution.
2. The parser basically checks the query for correct syntax and translates it into a conventional parse-tree (often called a query-tree) or some other internal representation.
3. The optimizer invokes the internal representation of the query as input so that a query plan or execution plan may be devised for retrieving the information that is required.
4. Heuristic optimization often includes making transformations to the query tree by moving operators up and down the tree so that the transformed tree is equivalent to the tree before the transformations.

6.6 Key Words

Query Processing, Heuristics, Query Optimisation

6.7 Self Assessment Questions

1. Discuss the following rules governing the manipulation of relational algebraic expressions?
2. Explain the algorithms used for the processing of join operation?
3. Explain how you could estimate costs while performing query processing?

6.8 References/Suggested Readings

1. Data Base Systems by C.J.Date
2. Data Base Management Systems by Alexis Leon, Mathews Leon
3. http://databases.about.com/library

MCA 202/MS 11

Author: Abhishek Taneja **Vetter: Dr. Pradeep Bhatia**

Lesson: Concurrency Control Techniques **Lesson No. : 07**

Structure

7.0 Objectives

At the end of this chapter the reader will be able to:

- Describe DBMS Techniques for concurrency control
- Describe and distinguish locking and deadlock
- Describe locking techniques for concurrency control based on time stamp ordering
- Describe multi version concurrency control techniques

7.1 Introduction

Concurrency in terms of databases means allowing multiple users to access the data contained within a database at the same time. If concurrent access is not managed by the Database Management System (DBMS) so that simultaneous operations don't interfere with one another problems can occur when various transactions interleave, resulting in an inconsistent database.

Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions. Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.

Concurrent execution of user programs is essential for good DBMS performance. Because disk accesses are frequent, and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently. Interleaving actions of different user programs can lead to inconsistency: e.g., check is cleared while account balance is being computed. DBMS ensures such problems don’t arise: users can pretend they are using a single-user system.

Define Transaction

A transaction is a sequence of read and write operations on data items that logically functions as one unit of work

- It should either be done entirely or not at all
- If it succeeds, the effects of write operations persist (commit); if it fails, no effects of write operations persist (abort)
- These guarantees are made despite concurrent activity in the system, and despite failures that may occur

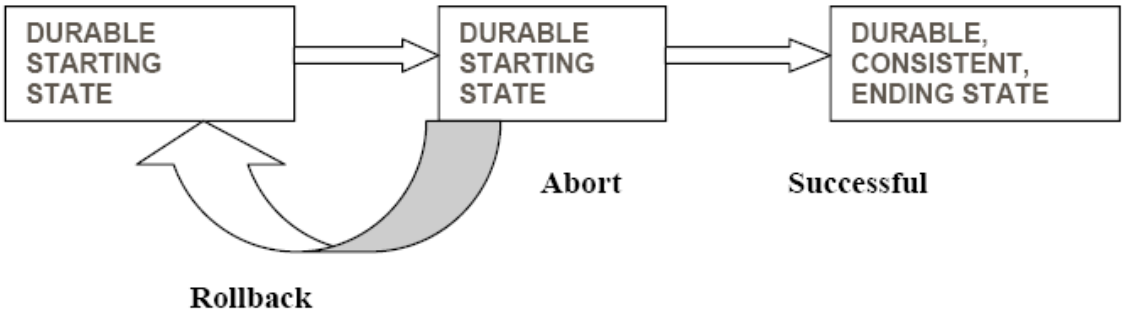


Figure 7.1

ACID Properties of Transaction

- **Atomic**

Process all of a transaction or none of it; transaction cannot be further subdivided (like an atom)

- **Consistent**

Data on all systems reflects the same state

- **Isolated**

Transactions do not interact/interfere with one another; transactions act as if they are independent

- **Durable**

Effects of a completed transaction are persistent

Concurrent Execution

You know there are good reasons for allowing concurrency:-

1. Improved throughput and resource utilization.

(THROUGHPUT = Number of Transactions executed per unit of time.)

The CPU and the Disk can operate in parallel. When a Transaction Read/Write the Disk another Transaction can be running in the CPU.

The CPU and Disk utilization also increases.

2. Reduced waiting time.

In a serial processing a short Transaction may have to wait for a long transaction to complete. Concurrent execution reduces the average response time; the average time for a Transaction to be completed.

7.1.1 What is concurrency control?

Concurrency control is needed to handle problems that can occur when transactions execute concurrently. The following are the concurrency issues:-

Lost Update: an update to an object by some transaction is overwritten by another interleaved transaction without knowledge of the initial update.

Lost Update Example:-

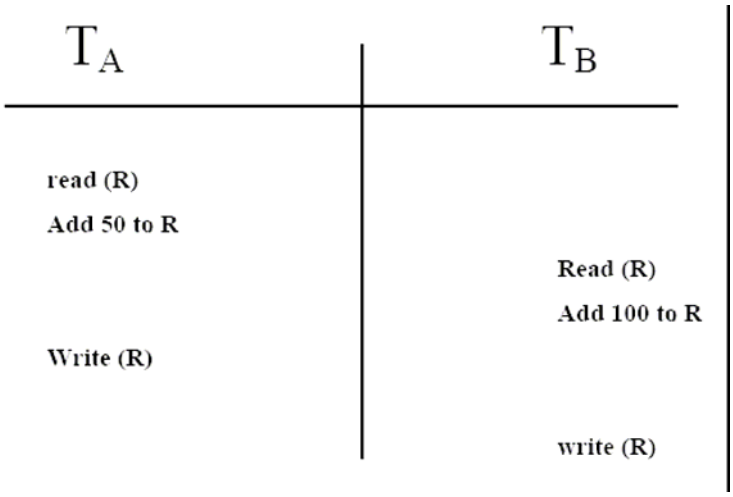


Figure 7.2

Transaction A's update is lost

Uncommitted Dependency: a transaction reads an object updated by another transaction that later falls.

Uncommitted Dependency Example:-

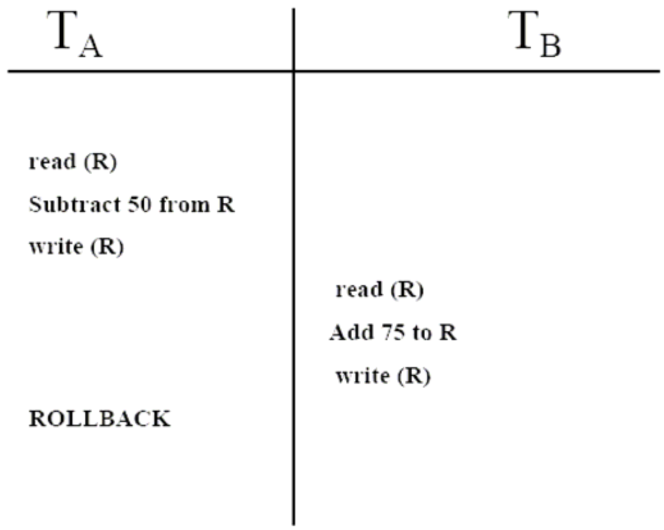


Figure 7.3

Transaction B reads an uncommitted value for R

Inconsistent Analysis: a transaction calculating an aggregate function uses some but not all updated objects of another transaction.

Inconsistent Analysis Example:-

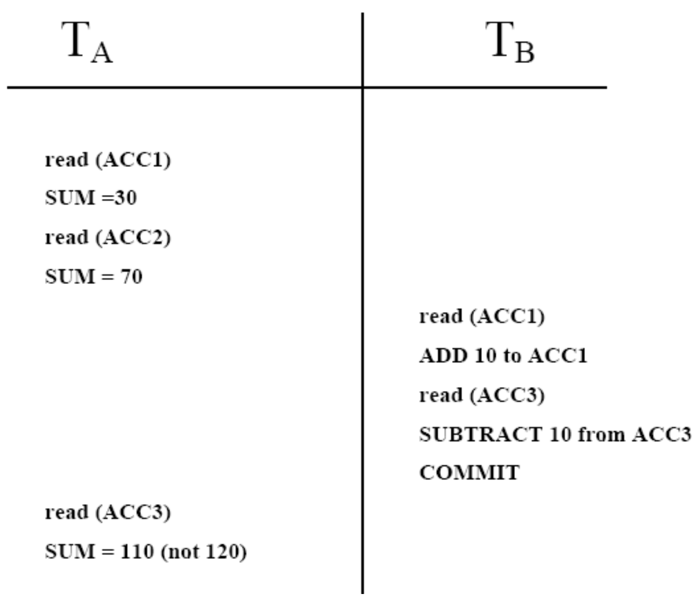


Figure 7.4

The value in SUM will be inconsistent

Main goals of Database Concurrency Control

- To point out problem areas in earlier performance analyses
- To introduce queuing network models to evaluate the baseline performance of transaction processing systems
- To provide insights into the relative performance of transaction processing systems
- To illustrate the application of basic analytic methods to the performance analysis of various concurrency control methods
- To review transaction models which are intended to relieve the effect of lock contention
- To provide guidelines for improving the performance of transaction processing systems due to concurrency control; and to point out areas for further investigation.

7.2 Transaction properties

To ensure data integrity the DBMS, should maintain the following transaction properties- atomicity, consistency, isolation and durability. These properties often referred to as **acid properties** an acronym derived from the first letter of the properties. In the last lecture we have introduced the above terms, now we will see their implementations.

We will consider the banking example to gain a better understanding of the acid properties and why are they important. We will consider a banking system that contains several accounts and a set of transactions that accesses and updates accounts. Access to a database is accomplished by two operations given below:-

1. Read(x)-This operation transfers the data item x from the database to a local buffer belonging to the transaction that executed the read operation
2. Write(x)-the write operation transfers the data item x from the local buffer of the transaction that executed the write operation to the database.

Now suppose that Ti is a transaction that transfers RS. 2000/- from account CA2090 to SB2359. This transaction is defined as follows:-

Ti:

Read(CA2090);

CA2090:=CA2090-2000;

Write (CA2090);

Read (SB2359);

SB2359:=SB2359+2000;

Write (SB2359);

We will now consider the acid properties...

Implementing Atomicity

Let's assume that before the transaction take place the balances in the account is Rs. 50000/- and that in the account SB2359 is Rs. 35000/-. Now suppose that during the execution of the transaction a failure(for example, a power failure) occurred that prevented the successful completion of the transaction. The failure occurred after the Write(CA2090); operation was executed, but before the execution of Write(SB2359); in this case the value of the accounts CA2090 and SB2359 are reflected in the database are Rs. 48,000/- and Rs. 35000/- respectively. The Rs. 200/- that we have taken from the account is lost. Thus the failure has created a problem. The state of the database no longer reflects a real state of the world that the database is supposed to capture. Such a state is called an inconsistent state. The database system should ensure that such inconsistencies are not visible in a database system. It should be noted that even during the successful execution of a transaction there exists points at which the system is in an inconsistent state. But the difference in the case of a successful transaction is that the period for which the database is in an inconsistent state is very short and once the transaction is over the system will be brought back to a consistent state. So if a transaction never started or is completed successfully, the inconsistent states would not be visible except during the execution of the transaction. This is the reason for the atomicity requirement. If the atomicity property provided all actions of the transaction are reflected in the database of none are. The mechanism of maintaining atomicity is as follows The DBMS keeps tracks of the old values of any data on which a transaction performs a Write and if the transaction does not complete its execution, old values are restored o make it appear as

though the transaction never took place. The transaction management component of the DBMS ensures the atomicity of each transaction.

Implementing Consistencies

The consistency requirement in the above eg is that the sum of CA2090 and SB2359 be unchanged by the execution of the transaction. Before the execution of the transaction the amounts in the accounts in CA2090 and SB2359 are 50,000 and 35,000 respectively. After the execution the amounts become 48,000 and 37,000. In both cases the sum of the amounts is 85,000 thus maintaining consistency. Ensuring the consistency for an individual transaction is the responsibility of the application programmer who codes the transaction. 2

Implementing the Isolation

Even if the atomicity and consistency properties are ensured for each transaction there can be problems if several transactions are executed concurrently. The different transactions interfere with one another and cause undesirable results. Suppose we are executing the above transaction Ti. We saw that the database is temporarily inconsistent while the transaction is being executed. Suppose that the transaction has performed the Write(CA2090) operation, during this time another transaction is reading the balances of different accounts. It checks the account CA2090 and finds the account balance at 48,000.

Suppose that it reads the account balance of the other account(account SB2359, before the first transaction has got a chance to update the account.

So the account balance in the account Sb2359 is 35000. After the second transaction has read the account balances, the first transaction reads the account balance of the account SB2359 and updates it to 37000.But here we are left with a problem. The first transaction has executed successfully and the database is back to a consistent state. But while it was in an inconsistent state, another transaction performed some operations(May be updated the total account balances). This has left the database in an inconsistent state even after both the transactions have been executed successfully. On solution to the situation(concurrent execution of transactions) is to execute the transactions serially- one after the

other. This can create many problems. Suppose long transactions are being executed first. Then all other transactions will have to wait in the queue. There might be many transactions that are independent(or that do not interfere with one another). There is no need for such transactions to wait in the queue. Also concurrent executions of transactions have significant performance advantages. So the DBMS have found solutions to allow multiple transactions to execute concurrency with out any problem. The isolation property of a transaction ensures that the concurrent execution of transactions result in a system state that is equivalent to a state that could have been obtained if the transactions were executed one after another. Ensuring isolation property is the responsibility of the concurrency-control component of the DBMS.

Implementing durability

The durability property guarantees that, once a transaction completes successfully, all updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution. We can guarantee durability by ensuring that either the updates carried out by the transaction have been written to the disk before the transaction completes or information about the updates that are carried out by the transaction and written to the disk are sufficient for the data base to reconstruct the updates when the data base is restarted after the failure. Ensuring durability is the responsibility of the recovery-management component of the DBMS

Picture

Transaction management and concurrency control components of a DBMS

Transaction States

Once a transaction is committed, we cannot undo the changes made by the transactions by rolling back the transaction. Only way to undo the effects of a committed transaction is to execute a compensating transaction. The creating of a compensating transaction can be quite complex and so the task is left to the user and it is not handled by the DBMS.

The transaction must be in one of the following states:-

1. active:- This is a initial state, the transaction stays in this state while it is executing
2. Partially committed:- The transaction is in this state when it has executed the final statement

3. Failed: - A transaction is in this state once the normal execution of the transaction cannot proceed.

4. Aborted: - A transaction is said to be aborted when the transaction has rolled back and the database is being restored to the consistent state prior to the start of the transaction.

5. Committed: - a transaction is in this committed state once it has been successfully executed and the database is transformed in to a new consistent state. Different transactions states arte given in following figure.

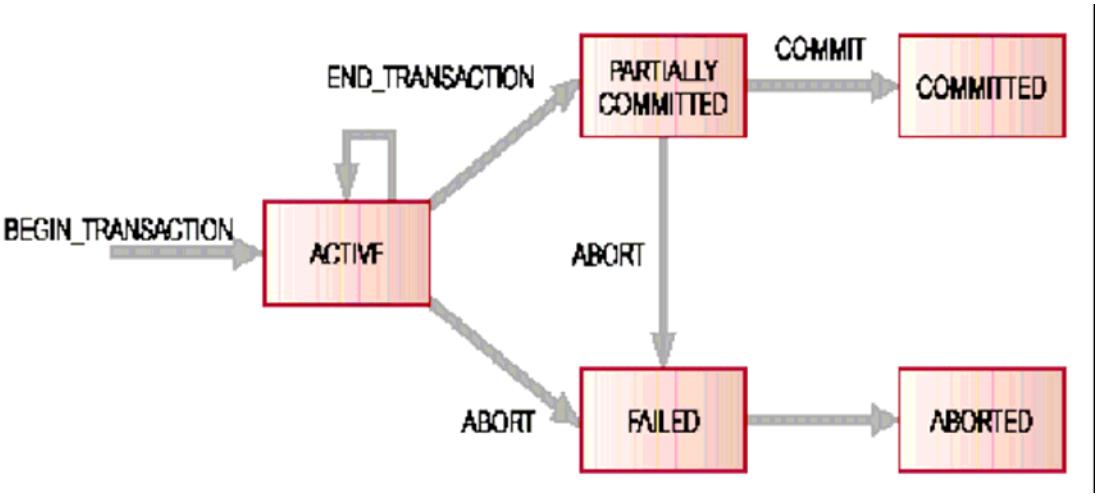


Figure 7.5 State Transition Diagram for a Transaction

7.3 Concurrency Control

As we have discussed this earlier, now we will talk about the concurrency control. Concurrency control in database management systems permits many users (assumed to be interactive) to access a database in a multi programmed environment while preserving the illusion that each user has sole access to the system. Control is needed to coordinate concurrent accesses to a DBMS so that the overall correctness of the database is maintained. For example, users A and B both may wish to read and update the same record in the database at about the same time. The relative timing of the two transactions may have an impact on the state of the database at the end of the transactions. The end result may be an inconsistent database.

Why Concurrent Control is needed?

• Several problems can occur when concurrent transactions execute in an uncontrolled manner.

– The lost update problem: This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of same database item incorrect.

– The temporary update (or dirty read) problem: This occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.

– The incorrect summary problem: If one transaction is calculating an aggregate function on a number of records while other transaction is updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

• Whenever a transaction is submitted to a DBMS for execution, the system must make sure that :

– All the operations in the transaction are completed successfully and their effect is recorded permanently in the database; or

– the transaction has no effect whatever on the database or on the other transactions in the case of that a transaction fails after executing some of operations but before executing all of them.

Following are the problems created due to the concurrent execution of the transactions:-

Multiple update problems

In this problem, the data written by one transaction (an update operation) is being overwritten by another update transaction. This can be illustrated using our banking example. Consider our account CA2090 that has Rs. 50000 balance in it. Suppose a transaction T1 is withdrawing RS. 10000 fro the account while another transaction T2 is depositing RS. 20000 to the account. If these transactions were executed serially (one after another), the final balance would be Rs. 60000, irrespective of the order in which the transactions are performed. In other words, if the transactions were performed serially, then the result would be the sameif T1 is performed first or T2 is performed first-

order is not important. But idf the transactions are performed concurrently, then depending on how the transactions are executed the results will vary. Consider the execution of the transactions given below

Sequence	T1	T2	Account Balance
01	Begin Transaction		50000
02	Read (CA2090)	Begin Transaction	50000
03	CA2090:=CA2090-10000	Read (CA2090)	50000
04	Write (CA2090)	CA2090:=CA2090+20000	40000
05	Commit	Write (CA2090)	70000
06		Commit	70000

Figure 7.6

Both transactions start nearly at the same time and both read the account balance of 50000. Both transactions perform the operations that they are supposed to perform-T1 will reduce the amount by 10000and will write the result to the data base; T2 will increase the amount by 20000 and will write the amount to the database overwriting the previous update. Thus the account balance will gain additional 10000 producing a wrong result. If T2 were to start execution first, the result would have been 40000 and the result would have been wrong again.

This situation could be avoided by preventing T2 from reading the value of the account balance until the update by T1 has been completed.

Incorrect Analysis Problem

Problems could arise even when a transaction is not updating the database. Transactions that read the database can also produce wrong result, if they are allowed to read the database when the database is in an inconsistent state. This problem is often referred to as dirty read or unrepeatable data. The problem of dirty read occurs when a transaction reads several values from the data base while another transactions are updating the values.

Consider the case of the transaction that reads the account balances from all accounts to find the total amount in various account. Suppose that there are other transactions, which are updating the account balances-either reducing the amount (withdrawals) or increasing

the amount (deposits). So when the first transaction reads the account balances and finds the totals, it will be wrong, as it might have read the account balances before the update in the case of some accounts and after the updates in other accounts. This problem is solved by preventing the first transaction (the one that reads the balances) from reading the account balances until all the transactions that update the accounts are completed.

Inconsistent Retrievals

Consider two users A and B accessing a department database simultaneously. The user A is updating the database to give all employees a 5% salary raise while user B wants to know the total salary bill of a department. The two transactions interfere since the total salary bill would be changing as the first user updates the employee records. The total salary retrieved by the second user may be a sum of some salaries before the raise and others after the raise. Such a sum could not be considered an acceptable value of the total salary (the value before the raise or after the raise would be).

A	Time	B
Read Employee 100	1	-
-	2	Sum = 0.0
Update Salary	3	-
-	4	Read Employee 100
Write Employee 100	5	-
-	6	Sum = Sum + Salary
Read Employee 101	7	-
-	8	Read Employee 101
Update Salary	9	-
-	10	Sum = Sum + Salary
Write Employee 101	11	-

Figure 7.7 An Example of Inconsistent Retrieval

The problem illustrated in the last example is called the inconsistent retrieval anomaly. During the execution of a transaction therefore, changes made by another transaction that has not yet committed should not be visible since that data may not be consistent.

Uncommitted Dependency

Consider the following situation:

A	Time	B
-	1	Read Q
-	2	-
-	3	Write A
Read Q	4	-
-	5	Read R
-	6	-
Write Q	7	-
-	8	Failure (rollback)
-	9	-

Figure 7.8 An Example of Uncommitted Dependency

Transaction A reads the value of Q that was updated by transaction B but was never committed. The result of Transaction A writing Q therefore will lead to an inconsistent state of the database. Also if the transaction A doesn't write Q but only reads it, it would be using a value of Q which never really existed! Yet another situation would occur if the roll back happens after Q is written by transaction A. The roll back would restore the old value of Q and therefore lead to the loss of updated Q by transaction A. This is called the

uncommitted dependency anomaly.

Serializability

Serializability is a given set of interleaved transactions is said to be serial sable if and only if it produces the same results as the serial execution of the same transactions Serializability is an important concept associated with locking. It guarantees that the work of concurrently executing transactions will leave the database state as it would have been if these transactions had executed serially. This requirement is the ultimate criterion for database consistency and is the motivation for the two-phase locking protocol, which dictates that no new locks can be acquired on behalf of a transaction after the DBMS releases a lock held by that transaction. In practice, this protocol generally means that locks are held until commit time.

Serializability is the classical concurrency scheme. It ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order. It assumes that all accesses to the database are done using read and write operations. A schedule is called ``correct" if we can find a serial schedule that is ``equivalent" to it. Given a set of transactions T1...Tn, two schedules S1 and S2 of these transactions are equivalent if the following conditions are satisfied:

Read-Write Synchronization: If a transaction reads a value written by another transaction in one schedule, then it also does so in the other schedule.

Write-Write Synchronization: If a transaction overwrites the value of another transaction in one schedule, it also does so in the other schedule.

These two properties ensure that there can be no difference in the effects of the two schedules

Serializable schedule

A schedule is serial if, for every transaction T participating the schedule, all the operations of T are executed consecutively in the schedule. Otherwise it is called non-serial schedule.

- Every serial schedule is considered correct; some nonserial schedules give erroneous results.

• A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions; a nonserial schedule which is not equivalent to any serial schedule is not serializable.

• The definition of two schedules considered “equivalent”:

– result equivalent: producing same final state of the database (is not used)

– conflict equivalent: If the order of any two conflicting operations is the same in both schedules.

– view equivalent: If each read operation of a transaction reads the result of the same write operation in both schedules and the write operations of each transaction must produce the same results.

• Conflict serializable: if a schedule S is conflict equivalent to some serial schedule. we can reorder the non-conflicting operations S until we form the equivalent serial schedule, and S is a serializable schedule.

• View Serializability: Two schedules are said to be view equivalent if the following three conditions hold. The same set of transactions participate in S and S'; and S and S' include the same operations of those transactions. A schedule S is said to be view serializable if it is view equivalent to a serial schedule.

7.4 Locking

In order to execute transactions in an interleaved manner it is necessary to have some form of concurrency control.

• This enables a more efficient use of computer resources.

• One method of avoiding problems is with the use of locks.

• When a transaction requires a database object it must obtain a lock.

Locking is necessary in a concurrent environment to assure that one process does not retrieve or update a record that is being updated by another process. Failure to use some controls (locking), would result in inconsistent and corrupt data.

Locks enable a multi-user DBMS to maintain the integrity of transactions by isolating a transaction from others executing concurrently. Locks are particularly critical in write-

intensive and mixed workload (read/write) environments, because they can prevent the inadvertent loss of data or Consistency problems with reads.

In addition to record locking, DBMS implements several other locking mechanisms to ensure the integrity of other data structures that provide shared I/O, communication among different processes in a cluster and automatic recovery in the event of a process or cluster failure.

Aside from their integrity implications, locks can have a significant impact on performance. While it may benefit a given application to lock a large amount of data (perhaps one or more tables) and hold these locks for a long period of time, doing so inhibits concurrency and increases the likelihood that other applications will have to wait for locked resources.

Locking Rules

You know there are various locking rules that are applicable when a user reads or writes a data to a database. The various locking rules are -

- Any number of transactions can hold S-locks on an item
- If any transaction holds an X-lock on an item, no other transaction may hold any lock on the item
- A transaction holding an X-lock may issue a write or a read request on the data item
- A transaction holding an S-lock may only issue a read request on the data item

7.5 Deadlock

You know there lies a threat while writing or reading data onto a database with the help of available resources, it is nothing but the deadlock condition.

In operating systems or databases, a situation in which two or more processes are prevented from continuing while each waits for resources to be freed by the continuation of the other. Any of a number of situations where two or more processes cannot proceed because they are both waiting for the other to release some resource.

A situation in which processes of a concurrent processor are waiting on an event which will never occur. A simple version of deadlock for a loosely synchronous environment arises when blocking reads and writes are not correctly matched. For example, if two nodes both execute blocking writes to each other at the same time, deadlock will occur

since neither write can complete until a complementary read is executed in the other node.

7.6 Locking Techniques For Concurrency Control Based On Time Stamp Ordering

The timestamp method for concurrency control does not need any locks and therefore there are no deadlocks. Locking methods generally prevent conflicts by making transaction to walk. Timestamp methods do not make the transactions wait. Transactions involved in a conflict are simply rolled back and restarted. A timestamp is a unique identifier created by the DBMS that indicates the relative starting time of a transaction. Timestamps are generated either using the system clock (generating a timestamp when the transaction starts to execute) or by incrementing a logical counter every time a new transaction starts.

Time stamping is the concurrency control protocol in which the fundamentals goal is to order transactions globally in such away that older transactions get priority in the event of a conflict. In the Timestamping method, if a transaction attempts to read or write a data item, then a read or write operation is allowed only if the last update on that data item was carried out by an older transaction. Otherwise the transaction requesting the read or write is restarted and given a new timestamp to prevent it from continually aborting and restarting. If the restarted transaction is not allowed a new timestamp and is allowed a new timestamp and is allowed to retain the old timestamp, it will never be allowed to perform the read or write, because by that some other transaction which has a newer timestamp than the restarted transaction might not be to commit due to younger transactions having already committed.

In addition to the timestamp for the transactions, data items are also assigned timestamps. Each data item contains a read-timestamp and write-timestamp. The read-timestamp contains the timestamp of the last transaction that read the item and the write-timestamp contains the timestamp of the last transaction that updated the item. For a transaction T the timestamp ordering protocol works as follows:

- Transactions T requests to read the data item ‘X’ that has already been updated by a younger (later or one with a greater timestamp) transaction. This means that an earlier transactions is trying to read a data item that has been updated by a later transaction T is too late to read the previous outdated value and any other values it has acquired are

likely to be inconsistent with the updated value of the data item. In this situation, the transaction T is aborted and restarted with a new timestamp.

- In all other cases, the transaction is allowed to proceed with the read operation. The read-timestamp of the data item is updated with the timestamp of transaction T.
- Transaction t requests to write (update) the data item 'X' that has already been read by a younger (later or one with the greater timestamp) transaction. This means that the younger transaction is already using the current value of the data item and it would be an error to update it now. This situation occurs when a transaction is late in performing the write and a younger transaction has already read the old value or written a new one. In this case the transaction T is aborted and is restarted with a new timestamp.
- Transaction T asks to write the data item 'X' that has already been written by a younger transaction. This means that the transaction T is attempting to write an old or obsolete value of the data item. In this case also the transaction T is aborted and is restarted with a new timestamp.
- In all other cases the transaction T is allowed to proceed and the write-timestamp of the data item is updated with the timestamp of transaction T.

  The above scheme is called basic timestamp ordering. This scheme guarantees that the transactions are conflict serializable and the results are equivalent to a serial schedule in which the transactions are executed in chronological order by the timestamps. In other words, the results of a basic timestamps ordering scheme will be as same as when all the transactions were executed one after another without any interleaving.

One of the problems with basic timestamp ordering is that it does not guarantee recoverable schedules. A modification to the basic timestamp ordering protocol that relaxes the conflict Serializability can be used to provide greater concurrency by rejecting obsolete write operations. This extension is known as Thomas's write rule. Thomas's write rule modifies the checks for a write operation by transaction T as follows.

- When the transaction T requests to write the data item 'X' whose values has already been read by a younger transaction. This means that the order transaction (transaction T) is writing an obsolete value to the data item. In this case the write operation is

ignored and the transaction (transaction T) is allowed to continue as if the write were performed. This principle is called the 'ignore obsolete write rule'. This rule allows for greater concurrency.

- In all other cases the transactions T is allowed to proceed and the write-timestamp of transaction T.

7.7 Multiversion Concurrency Control Techniques (MVCC)

The aim of Multi-Version Concurrency is to avoid the problem of Writers blocking Readers and vice-versa, by making use of multiple versions of data. The problem of Writers blocking Readers can be avoided if Readers can obtain access to a previous version of the data that is locked by Writers for modification.

The problem of Readers blocking Writers can be avoided by ensuring that Readers do not obtain locks on data. Multi-Version Concurrency allows Readers to operate without acquiring any locks, by taking advantage of the fact that if a Writer has updated a particular record, its prior version can be used by the Reader without waiting for the Writer to Commit or Abort. In a Multi-version Concurrency solution, Readers do not block Writers, and vice versa. While Multi-version concurrency improves database concurrency, its impact on data consistency is more complex.

7.7.1 Requirements of Multi-Version Concurrency systems

As its name implies, multi-version concurrency relies upon multiple versions of data to achieve higher levels of concurrency. Typically, a DBMS offering multi-version concurrency (MVDB), needs to provide the following features:

1. The DBMS must be able to retrieve older versions of a row.

2. The DBMS must have a mechanism to determine which version of a row is valid in the context of a transaction.

Usually, the DBMS will only consider a version that was committed prior to the start of the transaction that is running the query. In order to determine this, the DBMS must know which transaction created a particular version of a row, and whether this transaction committed prior to the starting of the current transaction.

7.7.2 Approaches to Multi-Version Concurrency

There are essentially two approaches to multi-version concurrency. The first approach is to store multiple versions of records in the database, and garbage collect records when

they are no longer required. This is the approach adopted by PostgreSQL and Firebird/Interbase. The second approach is to keep only the latest version of data in the database, as in SVDB implementations, but reconstruct older versions of data dynamically as required by exploiting information within the Write Ahead Log. This is the approach taken by Oracle and MySQL/InnoDb.

7.8 Summary

- Concurrency in terms of databases means allowing multiple users to access the data contained within a database at the same time.
- A transaction is a sequence of read and write operations on data items that logically functions as one unit of work
- Concurrency control is needed to handle problems that can occur when transactions execute concurrently.
- To ensure data integrity the DBMS, should maintain the following transaction properties- atomicity, consistency, isolation and durability **(ACID)**.
- Concurrency control in database management systems permits many users (assumed to be interactive) to access a database in a multi programmed environment while preserving the illusion that each user has sole access to the system.
- **Serializability** is a given set of interleaved transactions is said to be serial sable if and only if it produces the same results as the serial execution of the same transactions Serializability is an important concept associated with locking.
- Locking is necessary in a concurrent environment to assure that one process does not retrieve or update a record that is being updated by another process.
- A **timestamp** is a unique identifier created by the DBMS that indicates the relative starting time of a transaction.

7.9 Key Words

Concurrency Control, Locking, Dead Lock, Time Stamp, ACID Properties, Transaction, Serializability

7.10 Self Assessment Questions

1. What do you mean by concurrency?

2. What is concurrency control?
3. Explain Transaction with live examples?
4. What are the ACID Properties of Transaction?
5. Why concurrency is needed?
6. What do you mean by locking?
7. What do you mean by deadlocks?
8. Explain the timestamp ordering protocol?
9. Explain the Timestamping control?

7.11 References/Suggested Readings

1. Date, C, J, Introduction to Database Systems, 7th edition 9
2. Silbershatz, Korth, Sudarshan, Database System Concepts 4th Edition.

MCA 202/MS 11

Author: Abhishek Taneja **Vetter: Dr. Pradeep Bhatia**

Lesson: Database RecoveryTechniques **Lesson No. : 8**

Structure

8.0 Objectives

At the end of this chapter the reader will be able to:

- Describe some of the techniques that can be used for database recovery from failures.
- Describe several recovery concepts, including write ahead logging, in-place versus shadow updates, and the process of rolling back (undoing) the effect of an incomplete or failed transaction.
- Describe the technique known as shadowing or shadow paging
- Describe techniques for recovery from catastrophic failure

8.1 Introduction

In this chapter we discuss some of the techniques that can be used for database recovery from failures. We have already discussed the different causes of failure, such as system crashes and transaction *errors*. We start Section 8.2 with an outline of a typical recovery procedures and a categorization of recovery algorithms, and then discuss several recovery concepts, including write ahead logging, in-place versus shadow updates, and the process of rolling back (undoing) the effect of an incomplete or failed transaction. In Section 8.3, we present recovery techniques based on *deferred update,* also known as the NO-UNDO/REDO technique. In Section 8.4, we discuss recovery techniques based on immediate update; these include the UNDO/REDO and UNDO/NO-REDO algorithms. We discuss the technique known as shadowing or shadow paging, which can be categorized as a NO-UNDO/NO-REDO algorithm in Section 8.5. Finally, techniques for recovery from catastrophic failure are discussed in Section 8.6. Our emphasis is on conceptually describing several different approaches to recovery.

8.2 Recovery Concepts

8.2.1 Recovery Outline and Categorization of Recovery Algorithms

Recovery from transaction failures usually means that the database is *restored* to the most recent consistent state just before the time of failure. To do this, the system must keep information about the changes that were applied to data items by the various transactions. This information is typically kept in the system log. A typical strategy for recovery may be summarized informally as follows:

1. If there is extensive damage to a wide portion of the database due to catastrophic failure, such as a disk crash, the recovery method restores a past copy of the database that was *backed up* to archival storage (typically tape) and reconstructs a more current state by reapplying or *redoing* the operations of committed transactions from the *backed up* log, up to the time of failure.

2. When the database is not physically damaged but has become inconsistent due to non catastrophic failures of types as we discussed in the previous chapter, the strategy is to reverse any changes that caused the inconsistency by *undoing* some operations. It may also be necessary to *redo* some operations in order to restore a consistent state of the

database, as we shall see. In this case we do not need a complete archival copy of the database. Rather, the entries kept in the online system log are consulted during recovery.

Conceptually, we can distinguish two main techniques for recovery from non catastrophic transaction failures: (l) deferred update and (2) immediate update. The deferred update techniques do not physically update the database on disk until *after* a transaction reaches its commit point; then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transaction workspace (or buffers). During commit, the updates are first recorded persistently in the log and then written to the database. If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet have been recorded in the database. Hence, deferred update is also known as the NO-UNDO/ REDO algorithm. We discuss this technique in Section 8.3.

In the immediate update techniques, the database may be updated by some operations of a transaction *before* the transaction reaches its commit point. However, these operations are typically recorded in the log on disk by force writing *before* they are applied to the database making recovery still possible. If a transaction fails after recording some changes in the database but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both *undo* and *redo* may be required during recovery. This technique, known as the UNDO/REDO algorithm, requires both operations, and is used most often in practice. A variation of the algorithm where all updates are recorded in the database before a transaction commits requires *undo* only, so it is known as the UNDO/NO-REDO algorithm. We discuss these techniques in Section 8.4.

8.2.2 Caching (Buffering) of Disk Blocks

The recovery process is often closely intertwined with operating system functions-in particular, the buffering and caching of disk pages in main memory. Typically, one or more diskpages that include the data items to be updated are cached into main memory buffers and then updated in memory before being written back to disk. The caching of disk pages is traditionally an operating system function, but because of its importance to the efficiency of recovery procedures, it is handled by the DBMS by calling low-level

operating systems routines. In general, it is convenient to consider recovery in terms of the database disk pages (blocks). Typically a collection of in-memory buffers, called the DBMS cache, is kept under the control of the DBMS for the purpose of holding these buffers. A directory for the cache is used to keep track of which database items are in the buffers.' This can be a table of <disk page address, buffer location> entries. When the DBMS requests action on some item, it first checks the cache directory to determine whether the disk page containing the item is in the cache. If it is not, then the item must be located on disk, and the appropriate disk pages are copied into the cache. It may be necessary to replace (or flush) some of the cache buffers to make space available for the new item. Some page-replacement strategy from operating systems, such as least recently used (LRU) or first-in-first-out (FIFO), can be used to select the buffers for replacement. Associated with each buffer in the cache is a dirty bit, which can be included in the directory entry, to indicate whether or not the buffer has been modified. When a page is first read from the database disk into a cache buffer, the cache directory is updated with the new disk page address, and the dirty bit is set to a (zero). As soon as the buffer is modified, the dirty bit for the corresponding directory entry is set to 1 (one). When the buffer contents are replaced (flushed) from the cache, the contents must first be written back to the corresponding disk page *only if its dirty bit* is 1. Another bit, called the pin-unpin bit, is also needed-a page in the cache is pinned (bit value 1 (one» if it cannot be written back to disk as yet.

Two main strategies can be employed when flushing a modified buffer back to disk. The first strategy, known as in-place updating, writes the buffer back to the *same original disk location,* thus overwriting the old value of any changed data items on disk, Hence, a single copy of each database disk block is maintained. The second strategy, known as shadowing, writes an updated buffer at a different disk location, so multiple versions of data items can be maintained. In general, the old value of the data item before updating is called the before image (BFIM), and the new value after updating is called the after image

(AFIM). In shadowing, both the BFIM and the AFIM can be kept on disk; hence, it is not strictly necessary to maintain a log for recovering. We briefly discuss recovery based on shadowing in Section 8.5.

8.2.3 Write-Ahead Logging, Steal/No-Steal, and Force/No-Force

When in-place updating is used, it is necessary to use a log for recovery. In this case, the recovery mechanism must ensure that the BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM in the database on disk. This process is generally known as write-ahead logging. Before we can describe a protocol for write-ahead logging, we need to distinguish between two types of log entry information included for a write command: (1) the information needed for UNDO and (2) that needed for REDO. A REDO type log entry includes the new value (AFIM) of the item written by the operation since this is needed to *redo* the effect of the operation from the log (by setting the item value in the database to its AFIM). The UNDO-type log entries include the old value (BFIM) of the item since this is needed to *undo* the effect of the operation from the log (by setting the item value in the database back to its BFIM). In an UNDO/REDO algorithm, both types of log entries are combined. In addition, when cascading rollback is possible, read_item entries in the log are considered to be UNDO-type entries.

As mentioned, the DBMS cache holds the cached database disk blocks, which include not only *data blocks* but also *index blocks* and *log blocks* from the disk. When a log record is written, it is stored in the current log block in the DBMS cache. The log is simply a sequential (append-only) disk file and the DBMS cache may contain several log blocks (for example, the last n log blocks) that will be written to disk. When an update to a data block-stored in the DBMS cache-is made, an associated log record is written to the last log block in the DBMS cache. With the write-ahead logging approach, the log blocks that contain the associated log records for a particular data block update must first be written to disk before the data block itself can be written back to disk. Standard DBMS recovery terminology includes the terms steal/no-steal and force/no force, which specify when a page from the database can be written to disk from the cache:

1. If a cache page updated by a transaction *cannot* be written to disk before the transaction commits, this is called a no-steal approach. The pin-unpin bit indicates if a page cannot be written back to disk. Otherwise, if the protocol allows writing an updated buffer *before* the transaction commits, it is called steal. Steal is used when the DBMS cache (buffer) manager needs a buffer frame for another transaction and the buffer

manager replaces an existing page that had been updated but whose transaction has not committed.

2. If all pages updated by a transaction are immediately written to disk when the transaction commits, this is called a force approach. Otherwise, it is called no-force. The deferred update recovery scheme in Section 8.3 follows a *no-steal* approach.

However, typical database systems employ a *steal/no-force* strategy. The advantage of steal is that it avoids the need for a very large buffer space to store all updated pages in memory. The advantage of no-force is that an updated page of a committed transaction may still be in the buffer when another transaction needs to update it, thus eliminating the I/O cost to read that page again from disk. This may provide a substantial saving in the number of I/O operations when a specific page is updated heavily by multiple transactions. To permit recovery when in-place updating is used, the appropriate entries required for recovery must be permanently recorded in the logon disk before changes are applied to the database. For example, consider the following write-ahead logging (WAL) protocol for a recovery algorithm that requires both UNDO and REDO:

1. The before image of an item cannot be overwritten by its after image in the database on disk until all UNDO-type log records for the updating transaction-up to this point in time-have been force-written to disk.

2. The commit operation of a transaction cannot be completed until all the REDO-type and UNDO-type log records for that transaction have been force-written to disk. To facilitate the recovery process, the DBMS recovery subsystem may need to maintain a number of lists related to the transactions being processed in the system. These include a list for active transactions that have started but not committed as yet, and it may also include lists of all committed and aborted transactions since the last checkpoint (see next section). Maintaining these lists makes the recovery process more efficient.

8.1.4 Checkpoints in the System log and Fuzzy Checkpointing

Another type of entry in the log is called a checkpoint. A [checkpoint] record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified. As a consequence of this, all transactions that have their [commit, T] entries in the log before a [checkpoint] entry do not need to have

their WRITE operations *redone* in case of a system crash, since all their updates will be recorded in the database on disk during check pointing.

The recovery manager of a DBMS must decide at what intervals to take a checkpoint. The interval may be measured in time-say, every m minutes-or in the number *t* of committed transactions since the last checkpoint, where the values of m or *t* are system parameters. Taking a checkpoint consists of the following actions:

1. Suspend execution of transactions temporarily.
2. Force-write all main memory buffers that have been modified to disk.
3. Write a [checkpoint] record to the log, and force-write the log to disk.
4. Resume executing transactions.

As a consequence of step 2, a checkpoint record in the log may also include additional information, such as a list of active transaction ids, and the locations (addresses) of the first and most recent (last) records in the log for each active transaction. This can facilitate undoing transaction operations in the event that a transaction must be rolled back.

The time needed to force-write all modified memory buffers may delay transaction processing because of step 1. To reduce this delay, it is common to use a technique called fuzzy checkpointing in practice. In this technique, the system can resume transaction processing after the [checkpoint] record is written to the log without having to wait for step 2 to finish. However, until step 2 is completed, the previous [checkpoint] record should remain valid. To accomplish this, the system maintains a pointer to the valid checkpoint, which continues to point to the previous [checkpoint] record in the log. Once step 2 is concluded, that pointer is changed to point to the new checkpoint in the log.

8.2.5 Transaction Rollback

If a transaction fails for whatever reason after updating the database, it may be necessary to roll back the transaction. If any data item values have been changed by the transaction and written to the database, they must be restored to their previous values (BFIMs). The undotype log entries are used to restore the old values of data items that must be rolled back. If a transaction T is rolled back, any transaction S that has, in the interim, read the value of some data item X written by T must also be rolled back. Similarly, once S is rolled back, any transaction R that has read the value of some data item Y written by S

must also be rolled back; and so on. This phenomenon is called cascading rollback, and can occur when the recovery protocol ensures *recoverable* schedules but does not ensure *strict* or *cascadeless* schedules. Cascading rollback, understandably, can be quite complex and time-consuming. That is why almost all recovery mechanisms are designed such that cascading rollback is *never required.*

Figure 8.1 shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown in Figure 8.1a. Figure 8.1b shows the system log at the point of a system crash for a particular execution schedule of these transactions. The values of data items A, B, C, and 0, which are used by the transactions, are shown to the right of the system log entries. We assume that the original item values, shown in the first line, are A = 30, B = 15, C = 40, and 0 = 20. At the point of system failure, transaction T_3 has not reached its conclusion and must be rolled back. The WRITE operations of T_3 , marked by a single * in Figure 8.1b, are the T_3 operations that are undone during transaction rollback. Figure 8.1c graphically shows the operations of the different transactions along the time axis.

We must now check for cascading rollback. From Figure 8.1c we see that transaction T_2 reads the value of item B that was written by transaction T_3; this can also be determined by examining the log. Because T_3 is rolled back, T_2 must now be rolled back, too. The WRITE operations of T_2, marked by ** in the log, are the ones that are undone. Note that only write_item operations need to be undone during transaction rollback; read_item operations are recorded in the log only to determine whether cascading rollback of additional transactions is necessary.

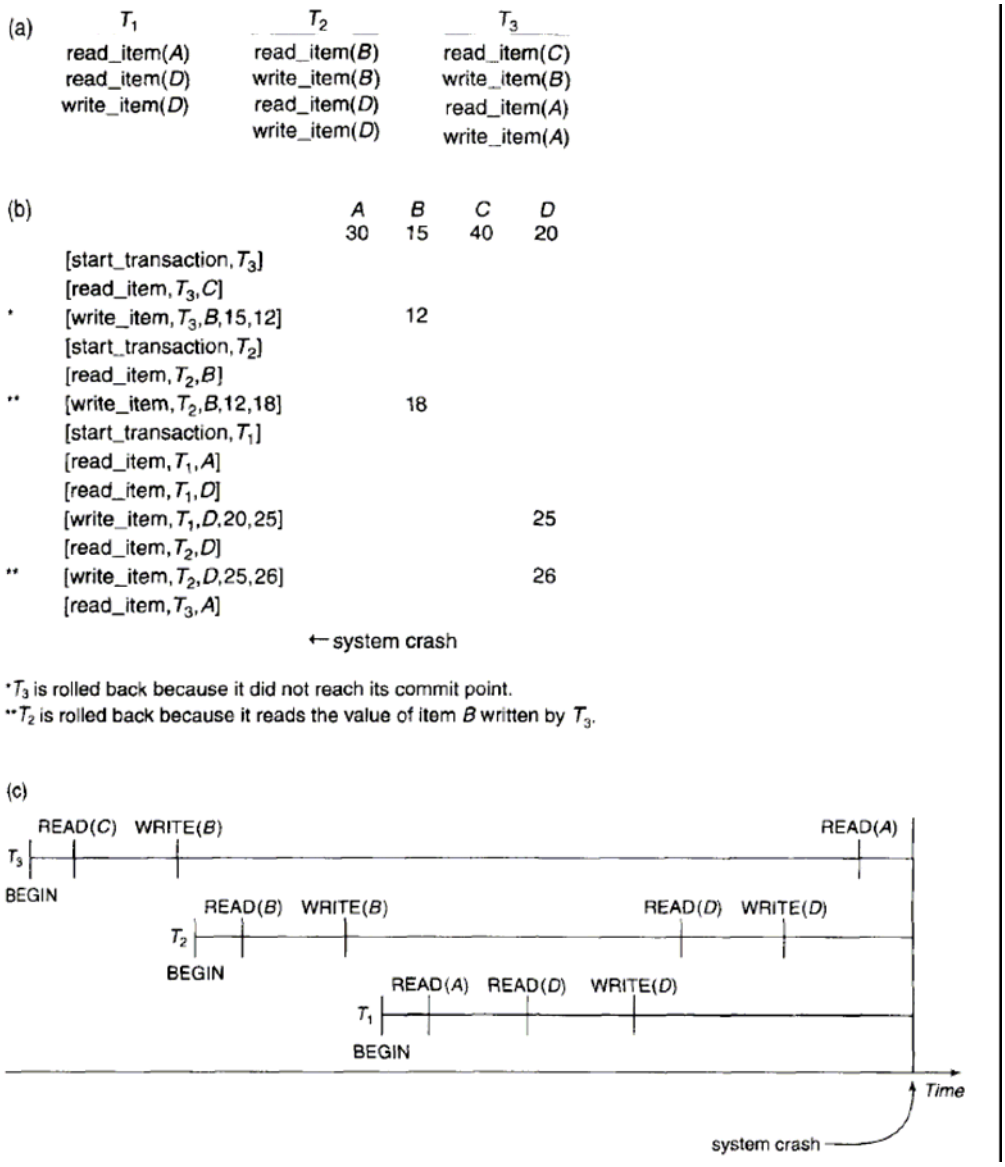


Figure 8.1 Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

In practice, cascading rollback of transactions is *never* required because practical recovery methods guarantee cascadeless or strict schedules. Hence, there is also no need to record any read_item operations in the log, because these are needed only for determining cascading rollback.

8.3 Recovery Techniques Based On Deferred Update

The idea behind deferred update techniques is to defer or postpone any actual updates to the database until the transaction completes its execution successfully and reaches its

commit point. During transaction execution, the updates are recorded only in the log and in the cache buffers. After the transaction reaches its commit point and the log is force-written to disk, the updates are recorded in the database. If a transaction fails before reaching its commit point, there is no need to undo any operations, because the transaction has not affected the database on disk in any way. Although this may simplify recovery, it cannot be used in practice unless transactions are short and each transaction changes few items. For other types of transactions, there is the potential for running out of buffer space because transaction changes must be held in the cache buffers until the commit point.

We can state a typical deferred update protocol as follows:

1. A transaction cannot change the database on disk until it reaches its commit point.
2. A transaction does not reach its commit point until all its update operations are recorded in the log *and* the log is force-written to disk.

Notice that step of this protocol is a restatement of the write-ahead logging (WAL) protocol. Because the database is never updated on disk until after the transaction commits, there is never a need to UNDO any operations. Hence, this is known as the **NO UNDO/ REDO recovery algorithm.** REDO is needed in case the system fails after a transaction commits but before all its changes are recorded in the database on disk. In this case, the transaction operations are redone from the log entries.

Usually, the method of recovery from failure is closely related to the concurrency control method in multi user systems. First we discuss recovery in single-user systems, where no concurrency control is needed, so that we can understand the recovery process independently of any concurrency control method. We then discuss how concurrency control may affect the recovery process.

8.3.1 Recovery Using Deferred Update in a Single-User Environment

In such an environment, the recovery algorithm can be rather simple. The algorithm RDU_S (Recovery using Deferred Update in a Single-user environment) uses a REDO procedure, given subsequently, for redoing certain write_item operations; it works as follows:

> PROCEDURE RDU_S: Use two lists of transactions: the committed transactions since the last checkpoint, and the active transactions (at most one transaction will

fall in this category, because the system is single-user). Apply the REDO operation to all the WRITE_ITEM operations of the committed transactions from the log in the order in which they were written to the log. Restart the active transactions.

The REDO procedure is defined as follows:

REDO(WRITE_OP): Redoing a write_item operation WRITE_OP consists of examining

its log entry [write_item, T, X, new_value] and setting the value of item X in the database to new_val ue, which is the after image (AFIM).

The REDO operation is required to be idempotent-that is, executing it over and over is equivalent to executing it just once. In fact, the whole recovery process should be idempotent. This is so because, if the system were to fail during the recovery process, the next recovery attempt might REDO certain write_item operations that had already been redone during the first recovery process. The result of recovery from a system crash *during recovery* should be the same as the result of recovering *when there is* no *crash during recovery* Notice that the only transaction in the active list will have had no effect on the database because of the deferred update protocol, and it is ignored completely by the recovery process because none of its operations were reflected in the database on disk. However, this transaction must now be restarted, either automatically by the recovery process or manually by the user.

Figure 8.2 shows an example of recovery in a single-user environment, where the first failure occurs during execution of transaction Tv as shown in Figure 8.2b. The recovery process will redo the [write_item, T1, D, 20] entry in the log by resetting the value of item D to 20 (its new value). The [write, T2, ...] entries in the log are ignored by the recovery process because T2 is not committed. If a second failure occurs during recovery from the first failure, the same recovery process is repeated from start to finish, with identical results.

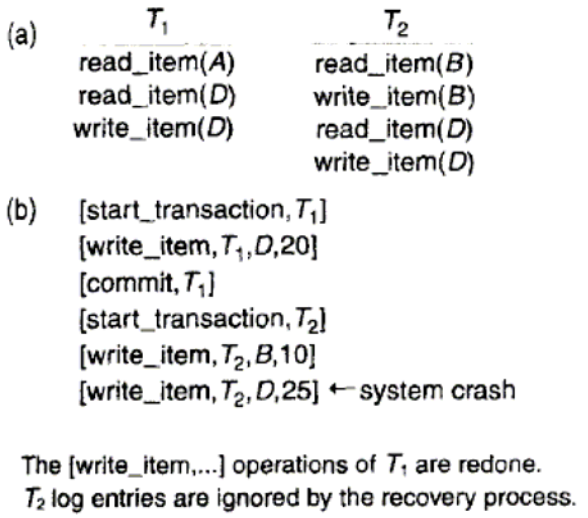


Figure 8.2 An example of recovery using deferred update in a single-user environment. (a) The READ and WRITE operations of two transactions. (b) The system log at the point of crash.

8.3.2 Deferred Update with Concurrent Execution in a Multi user Environment

For multi user systems with concurrency control, the recovery process may be more complex, depending on the protocols used for concurrency control. In many cases, the concurrency control and recovery processes are interrelated. In general, the greater the degree of concurrency we wish to achieve, the more time consuming the task of recovery becomes.

Consider a system in which concurrency control uses strict two-phase locking, so the locks on items remain in effect *until the transaction reaches its* commit *point.* After that, the locks can be released. This ensures strict and serializable schedules. Assuming that [checkpoint] entries are included in the log, a possible recovery algorithm for this case, which we call RDU_M (Recovery using Deferred Update in a Multi user environment), is given next. This procedure uses the REDO procedure defined earlier. PROCEDURE RDU_M (WITH CHECKPOINTS): Use two lists of transactions maintained by the system: the committed transactions T since the last checkpoint (commit list), and the active transactions T' (active list). REDO all the WRITE operations of the committed transactions from the log, in *the order in which they were written into the log.* The

transactions that are active and did not commit are effectively canceled and must be resubmitted.

Figure 8.3 shows a possible schedule of executing transactions. When the checkpoint was taken at time *t),* transaction T) had committed, whereas transactions T_3 and T_4 had not. Before the system crash at time t_2, T_3 and T_2 were committed but not T_4 and T_5. According to the RDU_M method, there is no need to redo the write_item operations of transaction T_1-or any transactions committed before the last checkpoint time *t).* Write_item operations of T_2 and T_3 must be redone, however, because both transactions reached

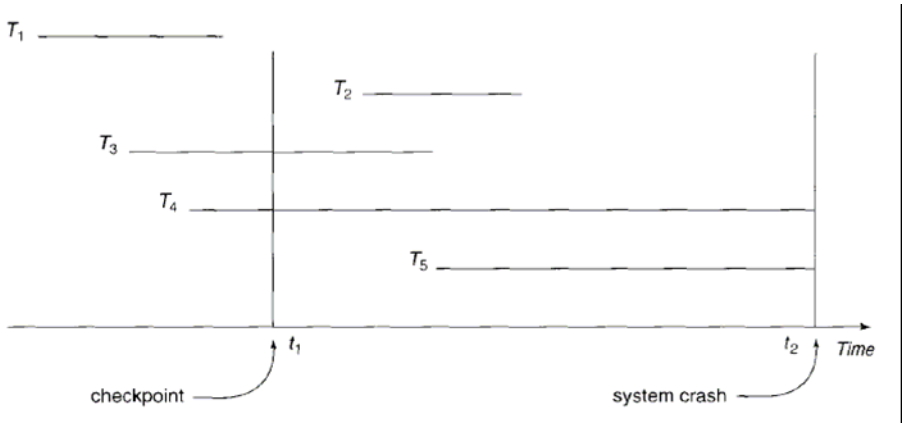


FIGURE 8.3 An example of recovery in a multiuser environment. their commit points after the last checkpoint.

Recall that the log is force-written before committing a transaction. Transactions T_4 and T_5 are ignored: They are effectively canceled or rolled back because none of their write_item operations were recorded in the database under the deferred update protocol. We will refer to Figure 8.3 later to illustrate other recovery protocols. We can make the NO-UNDO/REDO recovery algorithm *more efficient by* noting that, if a data item X has been updated-as indicated in the log entries-more than once by committed transactions since the last checkpoint, it is only necessary to REDO *the last update of* X from the log during recovery. The other updates would be overwritten by this last REDO in any case. In this case, we start from *the end of the log;* then, whenever an item is redone, it is added to a list of redone items. Before REDO is applied to an item, the list is checked; if the item appears on the list, it is not redone again, since its last value has already been

recovered. If a transaction is aborted for any reason (say, by the deadlock detection method), it is simply resubmitted, since it has not changed the database on disk. A drawback of the method described here is that it limits the concurrent execution of transactions because *all items remain locked until the transaction reaches its* commit *point.* In addition, it may

require excessive buffer space to hold all updated items until the transactions commit. The method's main benefit is that transaction operations *never need to be undone,* for two reasons:

1. A transaction does not record any changes in the database on disk until after it reaches its commit point-that is, until it completes its execution successfully. Hence, a transaction is never rolled back because of failure during transaction execution.

2. A transaction will never read the value of an item that is written by an uncommitted transaction, because items remain locked until a transaction reaches its commit point. Hence, no cascading rollback will occur. Figure 8.4 shows an example of recovery for a multi user system that utilizes the recovery and concurrency control method just described.

8.3.3 Transaction Actions That Do Not Affect the Database

In general, a transaction will have actions that do *not* affect the database, such as generating and printing messages or reports from information retrieved from the database. If a transaction fails before completion, we may not want the user to get these reports, since the transaction has failed to complete. If such erroneous reports are produced, part of the recovery process would have to inform the user that these reports are wrong, since the user may take an action based on these reports that affects the database. Hence, such reports should be generated only *after the transaction reaches its* commit *point.* A common method of dealing with such actions is to issue the commands that generate the reports but keep them as batch jobs, which are executed only after the transaction reaches its commit point. If the transaction fails, the batch jobs are canceled.

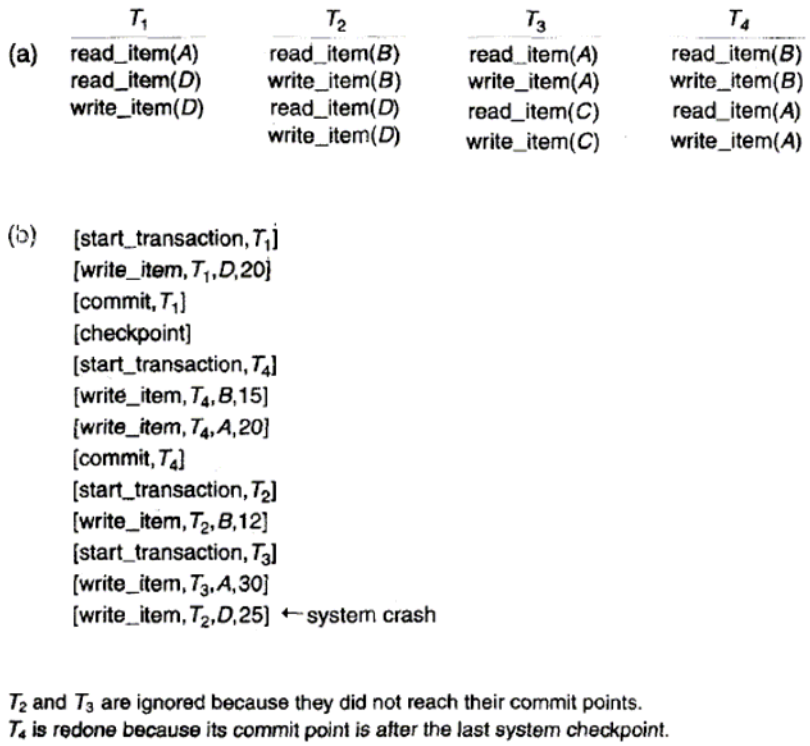


Figure 8.4 An example of recovery using deferred update with concurrent transactions. (a) The READ and WRITE operations of four transactions. (b) System log at the point of crash.

8.4 Recovery Techniques Based On Immediate Update

In these techniques, when a transaction issues an update command, the database can be updated "immediately," without any need to wait for the transaction to reach its commit point. In these techniques, however, an update operation must still be recorded in the log (on disk) *before* it is applied to the database-using the write-ahead logging protocol-so that we can recover in case of failure.

Provisions must be made for *undoing* the effect of update operations that have been applied to the database by a *failed transaction.* This is accomplished by rolling back the transaction and undoing the effect of the transaction's write_item operations. Theoretically, we can distinguish two main categories of immediate update algorithms. If the recovery technique ensures that all updates of a transaction are recorded in the database on disk *before the transaction* commits, there is never a need to REDO any operations of committed transactions.

This is called the UNDO/NO-REDO recovery algorithm. On the other hand, if the transaction is allowed to commit before all its changes are written to the database, we have the most general case, known as the UNDO/REDO recovery algorithm. This is also the most complex technique. Next, we discuss two examples of UNDO/REDO algorithms and leave it as an exercise for the reader to develop the UNDO/NO-REDO variation.

8.4.1 UNDO/REDO Recovery Based on Immediate Update in a Single-User Environment

In a single-user system, if a failure occurs, the executing (active) transaction at the time of failure may have recorded some changes in the database. The effect of all such operations must be undone. The recovery algorithm RIU_S (Recovery using Immediate Update in a Single-user environment) uses the REDO procedure defined earlier, as well as the UNDO procedure defined below.

PROCEDURE RIU_S

1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint and the active transactions (at most one transaction will fall in this category, because the system is single-user).
2. Undo all the write_item operations of the *active* transaction from the log, using the UNDO procedure described below.
3. Redo the write_item operations of the *committed* transactions from the log, in the order in which they were written in the log, using the REDO procedure described earlier.

The UNDO procedure is defined as follows:

UNDO(WRITE_OP): Undoing a write_item operation write_op consists of examining its log entry [write_item, T, X, old_va1ue, new_va1ue] and setting the value of item X in the database to old_va1ue which is the before image (BFIM). Undoing a number of write_item operations from one or more transactions from the log must proceed in the *reverse order* from the order in which the operations were written in the log.

8.4.2 UNDO/REDO Recovery Based on Immediate Update with Concurrent Execution

When concurrent execution is permitted, the recovery process again depends on the protocols used for concurrency control. The procedure RIU_M (Recovery using

Immediate Updates for a Multi user environment) outlines a recovery algorithm for concurrent transactions with immediate update. Assume that the log includes checkpoints and that the concurrency control protocol produces *strict schedules-as*, for example, the strict two phase locking protocol does. Recall that a strict schedule does not allow a transaction to read or write an item unless the transaction that last wrote the item has committed (or aborted and rolled back). However, deadlocks can occur in strict two-phase locking, thus requiring abort and UNDO of transactions. For a strict schedule, UNDO of an operation requires changing the item back to its old value (BFIM). PROCEDURE RIU_M

1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint and the active transactions.
2. Undo all the wri te_item operations of the *active* (uncommitted) transactions, using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log.
3. Redo all the wri te_item operations of the *committed* transactions from the log, in the order in which they were written into the log.

As we discussed in Section 8.3.2, step 3 is more efficiently done by starting from the *end of the log* and redoing only *the last update of each item* X. Whenever an item is redone, it is added to a list of redone items and is not redone again. A similar procedure can be devised to improve the efficiency of step 2.

8.5 Shadow Paging

This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, a log may be needed for the concurrency control method. Shadow paging considers the database to be made up of a number of fixed-size disk pages (or disk blocks)-say, n-for recovery purposes. A directory with n entries' is constructed, where the i^{th} entry points to the i^{th} database page on disk. The directory is kept in main memory if it is not too large, and all references-reads or writes-to database pages on disk go through it. When a transaction begins executing, the current directory-whose entries point to the most recent or current database pages on disk-is copied into a shadow directory. The shadow directory is then saved on disk while the current directory is used by the transaction.

During transaction execution, the shadow directory is *never* modified. When a write_ item operation is performed, a new copy of the modified database page is created, but the old copy of that page is *not overwritten.* Instead, the new page is written elsewhere-on some previously unused disk block. The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block. Figure 8.5 illustrates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory, and the new version by the current directory. To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory. The database thus is returned to its state

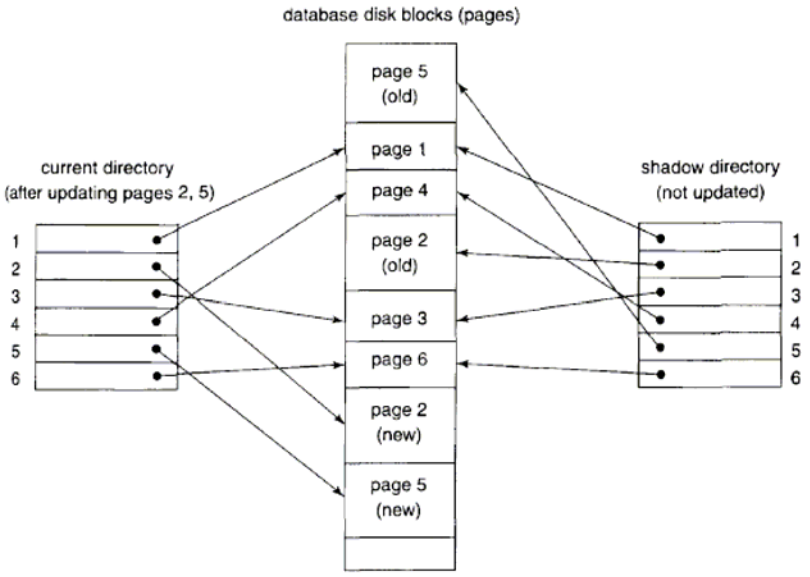


Figure 8.5 An example of shadow paging.

prior to the transaction that was executing when the crash occurred, and any modified pages are discarded. Committing a transaction corresponds to discarding the previous shadow directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery. In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. One disadvantage of shadow paging is

that the updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies. Furthermore, if the directory is large, the overhead of writing shadow directories to disk as transactions commit is significant. A further complication is how to handle garbage collection when a transaction commits. The old pages referenced by the shadow directory that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits. Another issue is that the operation to migrate between current and shadow directories must be implemented as an atomic operation.

8.6 Database Backup And Recovery From Catastrophic Failures

So far, all the techniques we have discussed apply to non catastrophic failures. A key assumption has been that the system log is maintained on the disk and is not lost as a result of the failure. Similarly, the shadow directory must be stored on disk to allow recovery when shadow paging is used. The recovery techniques we have discussed use the entries in the system log or the shadow directory to recover from failure by bringing the database back to a consistent state.

The recovery manager of a DBMS must also be equipped to handle more catastrophic failures such as disk crashes. The main technique used to handle such crashes is that of database backup. The whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes. In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted. To avoid losing all the effects of transactions that have been executed since the last backup, it is customary to back up the system log at more frequent intervals than full database backup by periodically copying it to magnetic tape. The system log is usually substantially smaller than the database itself and hence can be backed up more frequently. Thus users do not lose all transactions they have performed since the last database backup. All committed transactions recorded in the portion of the system log that has been backed up to tape can have their effect on the database redone. A new log is started after each database backup. Hence, to recover from disk failure, the database is first recreated on disk from its latest backup copy on tape. Following that, the effects of all the

committed transactions whose operations have been recorded in the backed-up copies of the system log are reconstructed.

8.7 Summary

1. Recovery from transaction failures usually means that the database is *restored* to the most recent consistent state just before the time of failure.
2. Conceptually, we can distinguish two main techniques for recovery from non catastrophic transaction failures: (l) deferred update and (2) immediate update.
3. The main goal of recovery is to ensure the atomicity property of a transaction.
4. Deferred update and immediate update. Deferred update techniques postpone any actual updating of the
   database on disk until a transaction reaches its commit point.
5. The deferred update approach, when used with certain concurrency control methods, is designed never to require transaction rollback, and recovery simply consists of redoing the operations of transactions committed after the last checkpoint from the log.
6. Shadow paging technique for recovery, which keeps track of old database pages by using a shadow directory. This technique, which is classified as NOUNDO/ NO-REDO, does not require a log in single-user systems but still needs the log for multiuser systems.
7. Recovery from catastrophic failures, which is typically done by backing up the database and the log to tape. The log can be backed up more frequently than the database, and the backup log can be used to redo operations starting from the last database backup.

8.8 Key Words

Database Recovery, Buffering, Rollback, Shadow Paging

8.9 Self Assessment Questions

1. Discuss the different types of transaction failures. What is meant by catastrophic failure?
2. Discuss the actions taken by the read_item and write_item operations on a database.

3. What is the system log used for? What are the typical kinds of entries in a system log? What are checkpoints, and why are they important?
4. What are transaction commit points, and why are they important?
5. How are buffering and caching techniques used by the recovery subsystem?
6. What are the before image (BFIM) and after image (AFIM) of a data item? What is the difference between in-place updating and shadowing, with respect to their handling of BFIM and AFIM?
7. What are UNDO-type and REDO-type log entries?
8. Describe the write-ahead logging protocol.
9. Identify three typical lists of transactions that are maintained by the recovery subsystem.
10. What is meant by transaction rollback? What is meant by cascading rollback?
11. Why do practical recovery methods use protocols that do not permit cascading rollback? Which recovery techniques do not require any rollback?
12. Discuss the UNDO and REDO operations and the recovery techniques that use each.
13. Discuss the deferred update technique of recovery.
14. What are the advantages and disadvantages of this technique? Why is it called the NO-UNDO/REDO method?
15. How can recovery handle transaction operations that do not affect the database, such as the printing of reports by a transaction?
16. Discuss the immediate update recovery technique in both single-user and multiuser environments. What are the advantages and disadvantages of immediate update?
17. What is the difference between the UNDO/REDO and the UNDO/NO-REDO algorithms for recovery with immediate update? Develop the outline for an UNDO/NO REDO algorithm.
18. Describe the shadow paging recovery technique. Under what circumstances does it not require a log?
19. Discuss how recovery from catastrophic failures is handled.

8.10 References/Suggested Readings

1. Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000
2. Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld
3. Elamasri R . and Navathe, S., Fundamentals of Database Systems (3rd Edition), Pearsson Education, 2000.

MCA 202/MS 11

Author: Abhishek Taneja **Vetter: Dr. Pradeep Bhatia**

Lesson: Distributed Databases and Client-Server Architectures Lesson No. : 9

Structure

9.0 Objectives

At the end of this chapter the reader will be able to:

- Describe distributed databases.
- Describe the design issues related to data fragmentation, replication, and distribution
- Distinguish between horizontal and vertical fragments of relations.
- Describe types of DDBMS
- Describe techniques used in distributed query processing,
- Describe client-server architecture concepts

9.1 Introduction

In this chapter we turn our attention to distributed databases (DDBs), distributed database management systems (DDBMSs), and how the client-server architecture is used as a platform for database application development. The DDB technology emerged as a merger of two technologies: (1) database technology, and (2) network and data communication technology. The latter has made tremendous strides in terms of wired and wireless technologies-from satellite and cellular communications and Metropolitan Area Networks (MANs) to the standardization of protocols like Ethernet, TCP/IP, and the Asynchronous Transfer Mode (ATM) as well as the explosion of the Internet. While early databases moved toward centralization and resulted in monolithic gigantic databases in the seventies and early eighties, the trend reversed toward more decentralization and autonomy of processing in the late eighties. With advances in distributed processing and distributed computing that occurred in the operating systems arena, the database research community did considerable work to address the issues of data distribution, distributed query and transaction processing, distributed database metadata management, and other topics, and developed many research prototypes. However, a full-scale comprehensive DDBMS that implements the functionality and techniques proposed in DDB research never emerged as a commercially viable product. Most major vendors redirected their efforts from developing a "pure" DDBMS product into developing systems based on client-server, or toward developing technologies for accessing distributed heterogeneous data sources.

Organizations, however, have been very interested in the *decentralization* of processing (at the system level) while achieving an *integration* of the information resources (at the logical level) within their geographically distributed systems of databases, applications, and users. Coupled with the advances in communications, there is now a general endorsement of the client-server approach to application development, which assumes many of the DDB issues.

In this chapter we discuss both distributed databases and client-server architectures in the development of database technology that is closely tied to advances in communications and network technology. Details of the latter are outside our scope. Section 9.2 introduces distributed database management and related concepts.

Detailed issues of distributed database design, involving fragmenting of data and distributing it over multiple sites with possible replication, are discussed in Section 9.3, Section 9.4 introduces different types of distributed database systems, including federated and multi database systems and highlights the problems of heterogeneity and the needs of autonomy in federated database systems, which will dominate for years to come. Sections 9.5 introduce distributed database query and finally in the last 9.6 section we discussed the architecture of distributed databases.

9.2 Distributed Database Concepts

Distributed databases bring the advantages of distributed computing to the database management domain. A distributed computing system consists of a number of processing elements, not necessarily homogeneous, that are interconnected by a computer network, and that cooperate in performing certain assigned tasks. As a general goal, distributed computing systems partition a big, unmanageable problem into smaller pieces and solve it efficiently in a coordinated manner. The economic viability of this approach stems from two reasons: (l) more computer power is harnessed to solve a complex task, and (2) each autonomous processing element can be managed independently and develop its own applications.

We can define a **distributed database** (DDB) as a collection of multiple logically interrelated databases distributed over a computer network, and a **distributed database management system (**DDBMS) as a software system that manages a distributed database while making the distribution transparent to the *user.* A collection of files stored at different nodes of a network and the maintaining of interrelationships among them via hyperlinks has become a common organization on the Internet, with files of Web pages. The common functions of database management, including uniform query processing and transaction processing, do *not* apply to this scenario yet. The technology is, however, moving in a direction such that distributed World Wide Web (WWW) databases will become a reality in the near future.

9.2.1 Parallel Versus Distributed Technology

Turning our attention to parallel system architectures, there are two main types of multiprocessor system architectures that are commonplace:

• *Shared memory (tightly coupled) architecture:* Multiple processors share secondary (disk) storage and also share primary memory.

• *Shared disk (loosely coupled) architecture:* Multiple processors share secondary (disk) storage but each has their own primary memory.

These architectures enable processors to communicate without the overhead of exchanging messages over a network: Database management systems developed using the above types of architectures are termed parallel database management systems rather than DDBMS, since they utilize parallel processor technology. Another type of multiprocessor architecture is called shared nothing architecture. In this architecture, every processor has its own primary and secondary (disk) memory, no common memory exists, and the processors communicate over a high-speed interconnection network (bus or switch). Although the shared nothing architecture resembles a distributed database computing environment, major differences exist in the mode of operation. In shared nothing multiprocessor systems, there is symmetry and homogeneity of nodes; this is not true of the distributed database environment where heterogeneity of hardware and operating system at each node is very common. Shared nothing architecture is also considered as an environment for parallel databases.

9.2.2 Advantages and Disadvantages of Distributed Databases

Advantages

- ***Matches distributed organizational model***
- ***Improved sharability and local autonomy***
- ***Improved availability***
- ***Improved reliability***
- ***Improved performance***
- ***Economics***
- ***Modular growth***

Disadvantages

- ***Complexity***
- ***Cost***
- ***Security***
- ***Lack of standards***
- ***Integrity control more difficult***
- ***Database design more complex***

9.3 Data Fragmentation, Replication, And Allocation Techniques For Distributed Database Design

In this section we discuss techniques that are used to break up the database into logical units, called **fragments**, which may be assigned for storage at the various sites. We also discuss the use of data **replication**, which permits certain data to be stored in more than one site, and the process of **allocating** fragments-or replicas of fragments-for storage at the various sites. These techniques are used during the process of **distributed database design**. The information concerning data fragmentation, allocation, and replication is stored in a global directory that is accessed by the DDBS applications as needed.

9.3.1 Data Fragmentation

In a DDB, decisions must be made regarding which site should be used to store which portions of the database. For now, we will assume that there is no *replication;* that is, each relation-or portion of a relation-is to be stored at only one site. We discuss replication and its effects later in this section. We also use the terminology of relational databases similar concepts apply to other data models. We assume that we are starting with a relational database schema and must decide on how to distribute the relations over the various sites.

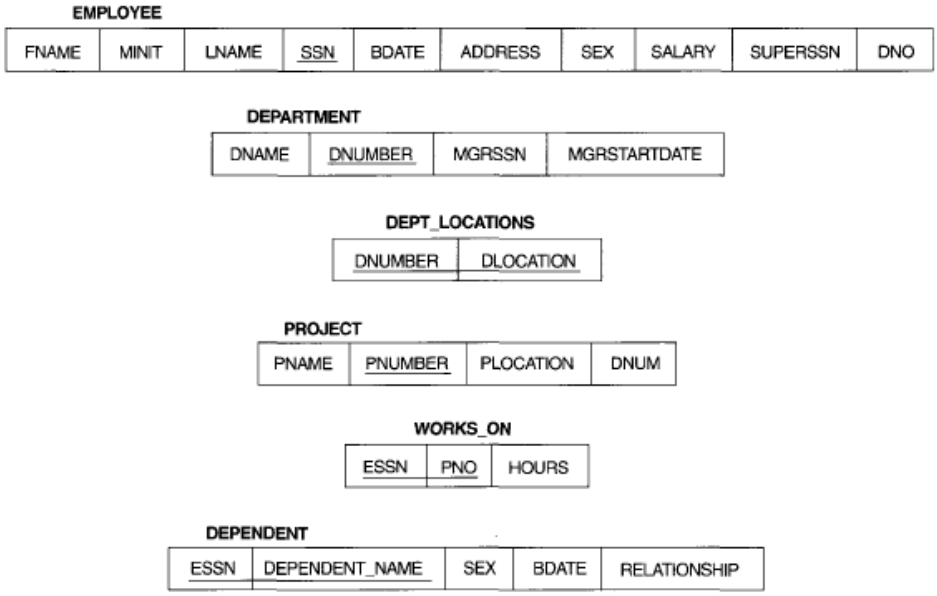
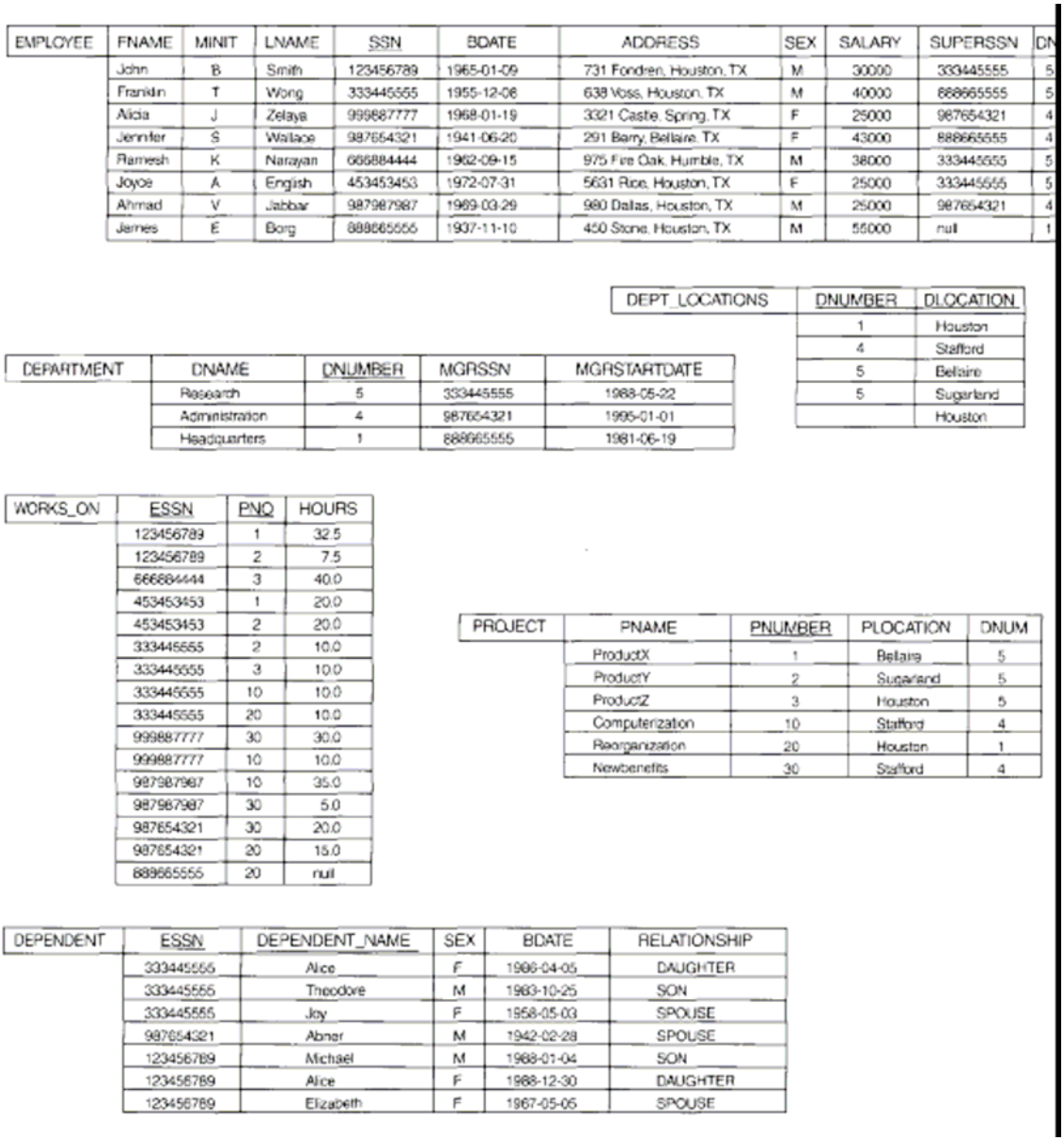


Figure 9.1 Schema diagram for the COMPANY relational database schema.

To illustrate our discussion, we use the relational database schema in Figure 9.1. Before we decide on how to distribute the data, we must determine the *logical units* of the database that are to be distributed. The simplest logical units are the relations themselves; that is, each *whole* relation is to be stored at a particular site. In our example, we must decide on a site to store each of the relations EMPLOYEE, DEPARTMENT, PROJECT, WORKS_ON, and DEPENDENT of Figure 9.1. In many cases, however, a relation can be divided into smaller logical units for distribution. For example, consider the company database shown in Figure 9.2, and assume there are three computer sites-one for each department in the company," We may want to store the database information relating to each department at the computer site for that department. A technique called *horizontal fragmentation* can be used to partition each relation by department.

Horizontal Fragmentation. A horizontal fragment of a relation is a subset of the tuples in that relation. The tuples that belong to the horizontal fragment are specified by a condition on one or more attributes of the relation. Often, only a single attribute is involved. For example, we may define three horizontal fragments on the EMPLOYEE relation of Figure 9.2 with the following conditions: (DNO = 5), (DNO = 4), and (DNO = l)----each fragment contains the EMPLOYEE tuples working for a particular department. Similarly, we may define three horizontal fragments for the PROJECT relation, with the conditions (DNUM = 5), (DNUM = 4), and (DNUM = I)--each fragment contains the PROJECT tuples controlled by a particular department. **Horizontal fragmentation** divides a relation "horizontally" by grouping rows to create subsets of tuples, where each subset has a certain logical meaning. These fragments can then be assigned to different sites in the distributed system. Derived horizontal fragmentation applies the partitioning of a



EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DN
	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

DEPT_LOCATIONS	DNUMBER	DLOCATION
	1	Houston
	4	Stafford
	5	Bellaire
	5	Sugarland
		Houston

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	5	333445555	1988-05-22
	Administration	4	987654321	1995-01-01
	Headquarters	1	888665555	1981-06-19

WORKS_ON	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0
	333445555	10	10.0
	333445555	20	10.0
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0
	987654321	20	15.0
	888665555	20	null

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
	ProductX	1	Bellaire	5
	ProductY	2	Sugarland	5
	ProductZ	3	Houston	5
	Computerization	10	Stafford	4
	Reorganization	20	Houston	1
	Newbenefits	30	Stafford	4

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
	333445555	Alice	F	1986-04-05	DAUGHTER
	333445555	Theodore	M	1983-10-25	SON
	333445555	Joy	F	1958-05-03	SPOUSE
	987654321	Abner	M	1942-02-28	SPOUSE
	123456789	Michael	M	1988-01-04	SON
	123456789	Alice	F	1988-12-30	DAUGHTER
	123456789	Elizabeth	F	1967-05-05	SPOUSE

Figure9.2 One possible database state for the COMPANY relational database schema.

primary relation (DEPARTMENT in our example) to other secondary relations (EMPLOYEE and PROJECT in our example), which are related to the primary via a foreign key. This way, related data between the primary and the secondary relations gets fragmented in the same way.

Vertical Fragmentation. Each site may not need all the attributes of a relation, which would indicate the need for a different type of fragmentation. Vertical fragmentation divides a relation "vertically" by columns. A vertical fragment of a relation keeps only

certain attributes of the relation. For example, we may want to fragment the EMPLOYEE relation into two vertical fragments. The first fragment includes personal information-NAME, BDATE, ADDRESS, and SEX-and the second includes work-related informarion-SSN, SALARY, SUPERSSN, DNO. This vertical fragmentation is not quite proper because, if the two fragments are stored separately, we cannot put the original employee tuples back together, since there is no *common attribute* between the two fragments. It is necessary to include the primary key or some candidate key attribute in *every* vertical fragment so that the full relation can be reconstructed from the fragments. Hence, we must add the SSN attribute to the personal information fragment.

Notice that each horizontal fragment on a relation R can be specified by a $\sigma_{Ci}(R)$ operation in the relational algebra. A set of horizontal fragments whose conditions CI, *C2,* ... , Cn include all the tuples in R-that is, every tuple in R satisfies (CI ORC2 OR... OR Cn)-is called a **complete horizontal fragmentation** of R. In many cases a complete horizontal fragmentation is also disjoint; that is, no tuple in R satisfies (C_i AND C_j) for any $i \neq j$. Our two earlier examples of horizontal fragmentation for the EMPLOYEE and PROJECT relations were both complete and disjoint. To reconstruct the relation R from a *complete* horizontal fragmentation, we need to apply the UNION operation to the fragments.

A vertical fragment on a relation R can be specified by a $\pi_{Li}(R)$ operation in the relational algebra. A set of vertical fragments whose projection lists L1, L2, ... , Ln include all the attributes in R but share only the primary key attribute of R is called a **complete vertical fragmentation** of R. In this case the projection lists satisfy the following two conditions:

• L1 U L2 U ... U Ln = ATTRS(R).

• $Li \cap Lj = PK(R)$ for any $i \neq j$, where ATTRS(R) is the set of attributes of R and PK(R) is the primary key of R.

To reconstruct the relation R from a *complete* vertical fragmentation, we apply the OUTER UNION operation to the vertical fragments (assuming no horizontal fragmentation is used). Notice that we could also apply a FULL OUTER JOIN operation and get the same result for a complete vertical fragmentation, even when some horizontal

fragmentation may also have been applied. The two vertical fragments of the EMPLDYEE relation with projection lists LI = {SSN, NAME, BDATE, ADDRESS, SEX} and L2 = {SSN, SALARY, SUPERSSN, DNO} constitute a complete vertical fragmentation of EMPLOYEE. Two horizontal fragments that are neither complete nor disjoint are those defined on the EMPLOYEE relation of Figure 9.1(a) by the conditions (SALARY> 50000) and (DNO = 4); they may not include all EMPLOYEE tuples, and they may include common tuples. Two vertical fragments that are not complete are those defined by the attribute lists L1 = {NAME, ADDRESS} and L2 = {SSN, NAME, SALARY}; these lists violate both conditions of a complete vertical fragmentation.

Mixed (Hybrid) Fragmentation ..We can intermix the two types of fragmentation, yielding a **mixed fragmentation**. For example, we may combine the horizontal and vertical fragmentations of the EMPLOYEE relation given earlier into a mixed fragmentation that includes six fragments. In this case the original relation can be reconstructed by applying UNION *and* OUTER UNION (or OUTER JOIN) operations in the appropriate order.

In general, a **fragment** of a relation R can be specified by a SELECT-PROJECT combination of operations $\pi_L(\sigma_C(R))$. If C = TRUE (that is, all tuples are selected) and L ≠ATTRS(R), we get a vertical fragment, and if C≠ TRUE and L = ATTRS(R), we get a horizontal fragment. Finally, if C ≠ TRUE and L ≠ ATTRS(R), we get a mixed fragment. Notice that a relation can itself be considered a fragment with C = TRUE and L = ATTRS(R). In the following discussion, the term *fragment* is used to refer to a relation or to any of the preceding types of fragments.

A **fragmentation schema** of a database is a definition of a set of fragments that includes *all* attributes and tuples in the database and satisfies the condition that the whole database can be reconstructed from the fragments by applying some sequence of OUTER UNION (or OUTER JOIN) and UNION operations. It is also sometimes useful-although not necessary-to have all the fragments be disjoint except for the repetition of primary keys among vertical (or mixed) fragments. In the latter case, all replication and distribution of fragments is clearly specified at a subsequent stage, separately from fragmentation.

An **allocation schema** describes the allocation of fragments to sites of the DDBS; hence, it is a mapping that specifies for each fragment the sitets) at which it is stored. If a

fragment is stored at more than one site, it is said to be replicated. We discuss data replication and allocation next.

9.3.2 Data Replication and Allocation

Replication is useful in improving the availability of data. The most extreme case is replication of the *whole database* at every site in the distributed system, thus creating a fully replicated distributed database. This can improve availability remarkably because the system can continue to operate as long as at least one site is up. It also improves performance of retrieval for global queries, because the result of such a query can be obtained locally from anyone site; hence, a retrieval query can be processed at the local site where it is submitted, if that site includes a server module. The disadvantage of full replication is that it can slow down update operations drastically, since a single logical update must be performed on every copy of the database to keep the copies consistent. This is especially true if many copies of the database exist. Full replication makes the concurrency control and recovery techniques more expensive than they would be if there were no replication, as we shall see in Section 9.6.

The other extreme from full replication involves having no replication-that is, each fragment is stored at exactly one site. In this case all fragments *must be* disjoint, except for the repetition of primary keys among vertical (or mixed) fragments. This is also called **non redundant allocation**.

Between these two extremes, we have a wide spectrum of **partial replication** of the data-that is, some fragments of the database may be replicated whereas others may not. The number of copies of each fragment can range from one up to the total number of sites in the distributed system. A special case of partial replication is occurring heavily in applications where mobile workers-such as sales forces, financial planners, and claims adjustors-carry partially replicated databases with them on laptops and personal digital assistants and synchronize them periodically with the server database. A description of the replication of fragments is sometimes called a **replication schema**.

Each fragment-or each copy of a fragment-must be assigned to a particular site in the distributed system. This process is called data distribution (or data allocation). The choice of sites and the degree of replication depend on the performance and availability goals of the system and on the types and frequencies of transactions submitted at each site. For

example, if high availability is required and transactions can be submitted at any site and if most transactions are retrieval only, a fully replicated database is a good choice. However, if certain transactions that access particular parts of the database are mostly submitted at a particular site, the corresponding set of fragments can be allocated at that site only. Data that is accessed at multiple sites can be replicated at those sites. If many updates are performed, it may be useful to limit replication. Finding an optimal or even a good solution to distributed data allocation is a complex optimization problem.

9.4 Types of Distributed Database Systems

The term distributed database management system can describe various systems that differ from one another in many respects. The main thing that all such systems have in common is the fact that data and software are distributed over multiple sites connected by some form of communication network. In this section we discuss a number of types of DDBMSs and the criteria and factors that make some of these systems different.

The first factor we consider is the **degree of homogeneity** of the DDBMS software. If all servers (or individual local DBMSs) use identical software and all users (clients) use identical software, the DDBMS is called **homogeneous**; otherwise, it is called **heterogeneous**. Another factor related to the degree of homogeneity is the **degree of local autonomy**. If there is no provision for the local site to function as a stand-alone DBMS, then the system has **no local autonomy**. On the other hand, if *direct access* by local transactions to a server is permitted, the system has some degree of local autonomy.

At one extreme of the autonomy spectrum, we have a DDBMS that "looks like" a centralized DBMS to the user. A single conceptual schema exists, and all access to the system is obtained through a site that is part of the DDBMS-which means that no local autonomy exists.

At the other extreme we encounter a type of DDBMS called a *federated* DDBMS (or a *multidatabase system).* In such a system, each server is an independent and autonomous centralized DBMS that has its own local users, local transactions, and DBA and hence has a very high degree of *local autonomy.* The term **federated database system** (FDBS) is used when there is some global view or schema of the federation of databases that is shared by the applications. On the other hand, a **multidatabase system** does not have a global schema and interactively constructs one as needed by the application. Both

systems are hybrids between distributed and centralized systems and the distinction we made between them is not strictly followed. We will refer to them as FDBSs in a generic sense.

In a heterogeneous FOBS, one server may be a relational DBMS, another a network DBMS, and a third an object or hierarchical DBMS; in such a case it is necessary to have a canonical system language and to include language translators to translate subqueries from the canonical language to the language of each server. We briefly discuss the issues affecting the design of FDBSs below.

Federated Database Management Systems Issues. The type of heterogeneity present in FDBSs may arise from several sources. We discuss these sources first and then point out how the different types of autonomies contribute to a semantic heterogeneity that must be resolved in a heterogeneous FOBS.

• *Differences* in *data models:* Databases in an organization come from a variety of data models including the so-called legacy models, the relational data model, the object data model, and even files. The modeling capabilities of the models vary. Hence, to deal with them uniformly via a single global schema or to process them in a single language is challenging. Even if two databases are both from the RDBMS environment, the same information may be represented as an attribute name, as a relation name, or as a value in different databases. This calls for an intelligent query-processing mechanism that can relate information based on metadata.

• *Differences* in *constraints:* Constraint facilities for specification and implementation vary from system to system. There are comparable features that must be reconciled in the construction of a global schema. For example, the relationships from ER models are represented as referential integrity constraints in the relational model. Triggers may have to be used to implement certain constraints in the relational model. The global schema must also deal with potential conflicts among constraints.

• *Differences* in *query languages:* Even with the same data model, the languages and their versions vary. For example, SQLhas multiple versions like SQL-89,

sQL-92, and SQL-99, and each system has its own set of data types, comparison operators, string manipulation features, and so on.

Semantic Heterogeneity. Semantic heterogeneity occurs when there are differences in the meaning, interpretation, and intended use of the same or related data. Semantic heterogeneity among component database systems (DBSs) creates the biggest hurdle in designing global schemas of heterogeneous databases. The design autonomy of component DBSs refers to their freedom of choosing the following design parameters, which in tum affect the eventual complexity of the FOBS:

- *The universe of discourse from which the data* is *drawn:* For example, two customer accounts, databases in the federation may be from United States and Japan with entirely different sets of attributes about customer accounts required by the accounting practices. Currency rate fluctuations would also present a problem. Hence, relations in these two databases which have identical names-CUSTOMER or ACCOUNT-may have some common and some entirely distinct information.
- *Representation and* naming: The representation and naming of data elements and the structure of the data model may be prespecified for each local database.
- *The understanding, meaning, and subjective interpretation of data.* This is a chief contributor to semantic heterogeneity.
- *Transaction and policy constraints:* These deal with serializability criteria, compensating transactions, and other transaction policies.
- *Derivation of summaries:* Aggregation, summarization, and other data-processing features and operations supported by the system.

Communication autonomy of a component DBS refers to its ability to decide whether to communicate with another component DBS. **Execution autonomy** refers to the ability of a component DBS to execute local operations without interference from external operations by other component DBSs and its ability to decide the order in which to execute them. The **association autonomy** of a component DBS implies that it has the

ability to decide whether and how much to share its functionality (operations it supports) and resources (data it manages) with other component DBSs. The major challenge of designing FDBSs is to let component DBSs interoperate while still providing the above types of autonomies to them.

9.5 Query Processing in Distributed Databases

We now give an overview of how a DDBMS processes and optimizes a query. We first discuss the communication costs of processing a distributed query; we then discuss a special operation, called a *semijoin,* that is used in optimizing some types of queries in a DDBMS.

9.5.1 Data Transfer Costs of Distributed Query Processing

In a distributed system, several additional factors complicate query processing. The first is the cost of transferring data over the network. This data includes intermediate files that are transferred to other sites for further processing, as well as the final result files that may have to be transferred to the site where the query result is needed. Although these costs may not be very high if the sites are connected via a high performance local area network, they become quite significant in other types of networks. Hence, DDBMS query optimization algorithms consider the goal of reducing the *amount of data transfer* as an optimization criterion in choosing a distributed query execution strategy. We illustrate this with two simple example queries. Suppose that the EMPLOYEE and DEPARTMENT relations of Figure 9.1 are distributed as shown in Figure 9.3. We will assume in this example that neither relation is fragmented. According to Figure 9.3, the size of the EMPLOYEE relation is 100 * 10,000 = 10^6 bytes, and the size of the DEPARTMENT relation is 35 * 100 = 3500 bytes. Consider the query Q: "For each

employee, retrieve the employee

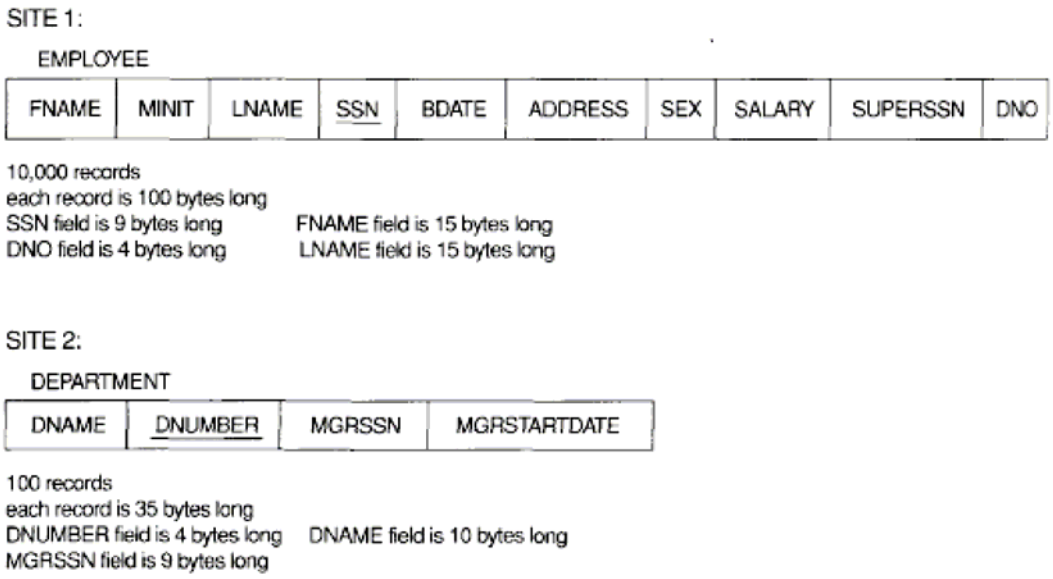


Figure 9.3 Example to illustrate *volume* of data transferred.

name and the name of the department for which the employee works." This can be stated as follows in the relational algebra:

$$Q: \pi_{FNAME, LNAME, DNAME}(EMPLOYEE \bowtie_{DNO=DNUMBER} DEPARTMENT)$$

The result of this query will include 10,000 records, assuming that every employee is related to a department. Suppose that each record in the query result is 40 *bytes long.* The query is submitted at a distinct site 3, which is called the result site because the query result is needed there. Neither the EMPLOYEE nor the DEPARTMENT relations reside at site 3.

There are three simple strategies for executing this distributed query:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case a total of 1,000,000 + 3500 = 1,003,500 bytes must be transferred.
2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is 40 * 10,000 = 400,000 bytes, so 400,000 + 1,000,000 =1,400,000 bytes must be transferred.

3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case 400,000 + 3500 = 403,500 bytes must be transferred.

If minimizing the amount of data transfer is our optimization criterion, we should choose strategy 3. Now consider another query *Q':* "For each department, retrieve the department name and the name of the department manager." This can be stated as follows in the relational algebra:

$$Q': \pi_{FNAME, LNAME, DNAME}(DEPARTMENT \bowtie_{MGRSSN=SSN} EMPLOYEE)$$

Again, suppose that the query is submitted at site 3. The same three 'strategies for executing query Q apply to *Q',* except that the result of Q' includes only 100 records, assuming that each department has a manager:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case a total of 1,000,000 + 3500 = 1,003,500 bytes must be transferred.

2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is 40 * 100 = 4000 bytes, so 4000 + 1,000,000 = 1,004,000 bytes must be transferred.

3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case 4000 + 3500 = 7500 bytes must be transferred.

Again, we would choose strategy 3-in this case by an overwhelming margin over strategies 1 and 2. The preceding three strategies are the most obvious ones for the case where the result site (site 3) is different from all the sites that contain files involved in the query (sites 1 and 2). However, suppose that the result site is site 2; then we have two simple strategies:

1. Transfer the EMPLOYEE relation to site 2, execute the query, and present the result to the user at site 2. Here, the same number of bytes-1 ,OOO,OOO-must be transferred for both Qand *Q'.*

2. Transfer the DEPARTMENT relation to site 1, execute the query at site 1, and send the result back to site 2. In this case 400,000 + 3500 = 403,500 bytes must be transferred for Q and 4000 + 3500 = 7500 bytes for *Q'*.

A more complex strategy, which sometimes works better than these simple strategies, uses an operation called **semijoin.** We introduce this operation and discuss distributed execution using semi joins next.

9.6 An Overview of Client-Server Architecture

As we pointed out in the chapter introduction, full-scale DDBMSs have not been developed to support all the types of functionalities that we discussed so far. Instead, distributed database applications are being developed in the context of the client-server architecttures. It is now more common to use a three-tier architecture, particular in Web applications. This architecture is illustrated in Figure 9.4. In the three-tier client-server architecture, the following three layers exist:

1. Presentation layer (client): This provides the user interface and interacts with the user. The programs at this layer present Web interfaces or forms to the client in order to interface with the application. Web browsers are often utilized, and the languages used include HTML, JAVA, JavaScript, PERL, Visual Basic, and so on. This layer handles user input, output, and navigation by accepting user commands and displaying the needed information, usually in the form of static or dynamic Web pages. The latter are employed when the interaction involves database access. When a Web interface is used, this layer typically communicates with the application layer via the HTTP protocol.
2. Application layer (business logic): This layer programs the application logic. For example, queries can be formulated based on user input from the client, or query results can be formatted and sent to the client for presentation. Additional application functionality can be handled at this layer, such as security checks, identity verification, and other functions. The application layer can interact with one or more databases or data sources as needed by connecting to the database using ODBC, JDBC, SQL/CLI or other database access techniques.
3. Database server: This layer handles query and update requests from the application layer, processes the requests, and send the results. Usually SQL is

used to access the database if it is relational or object-relational and stored database pro

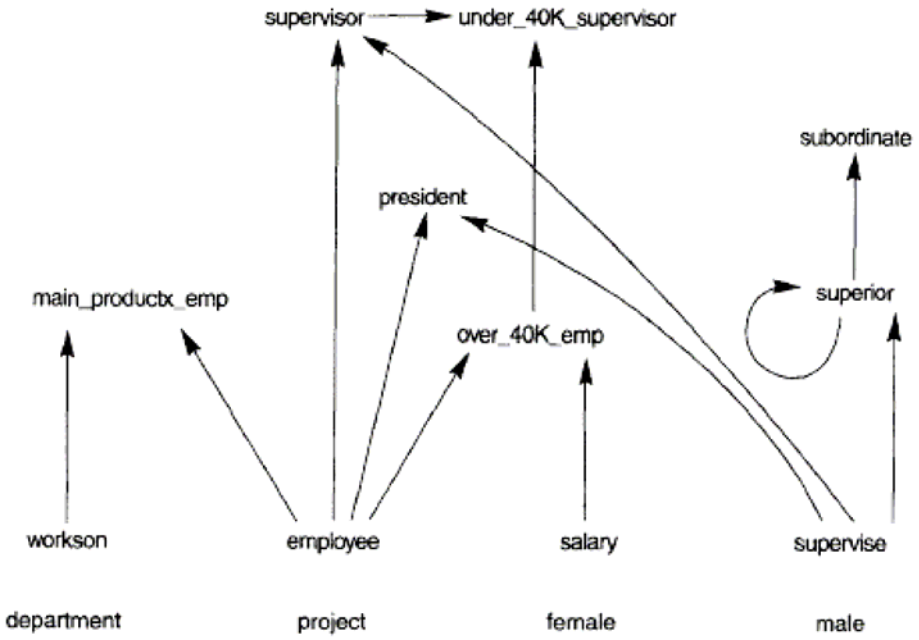


Figure 9.4 The three-tier client-server architecture.

cedures may also be invoked. Query results (and queries) may be formatted into XML when transmitted between the application server and the database server.

Exactly how to divide the DBMS functionality between client, application server, and database server may vary. The common approach is to include the functionality of a centralized DBMS at the database server level. A number of relational DBMS products have taken this approach, where an SQL server is provided. The application server must then formulate the appropriate SQL queries and connect to the database server when needed. The client provides the processing for user interface interactions. Since SQL is a relational standard, various SQL servers, possibly provided by different vendors, can accept SQL commands through standards such as ODBC, JDBC, SQL/CLI. In this architecture, the application server may also refer to a data dictionary that includes information on the distribution of data among the various SQL servers, as well as modules for decomposing a global query into a number of local queries that can be executed at the various sites. Interaction between application server and database server might proceed as follows during the processing of an SQL query:

1. The application server formulates a user query based on input from the client layer and decomposes it into a number of independent site queries. Each site query is sent to the appropriate database server site.

2. Each database server processes the local query and sends the results to the application server site. Increasingly, XML is being touted as the standard for data exchange so the database server may format the query result into XML before sending it to the application server.

3. The application server combines the results of the subqueries to produce the result of the originally required query, formats it into HTML or some other form accepted by the client, and sends it to the client site for display.

The application server is responsible for generating a distributed execution plan for a multisite query or transaction and for supervising distributed execution by sending commands to servers. These commands include local queries and transactions to be executed, as well as commands to transmit data to other clients or servers. Another function controlled by the application server (or coordinator) is that of ensuring consistency of replicated copies of a data item by employing distributed (or global) concurrency control techniques. The application server must also ensure the atomicity of global transactions by performing global recovery when certain sites fail.

If the DDBMS has the capabilty to *hide* the details of data distribution from the application server, then it enables the application server to execute global queries and transactions as though the database were centralized, without having to specify the sites at which the data referenced in the query or transaction resides. This property is called distribution transparency. Some DDBMSs do not provide distribution transparency, instead requiring that applications be aware of the details of data distribution.

9.7 Summary

1. Distributed databases bring the advantages of distributed computing to the database management domain.
2. We can define a **distributed database** (DDB) as a collection of multiple logically interrelated databases distributed over a computer network, and a **distributed database management system (**DDBMS) as a software system that manages a distributed database while making the distribution transparent to the *user.*

3. The techniques that are used to break up the database into logical units, called **fragments.**
4. Data **replication**, which permits certain data to be stored in more than one site
5. Vertical fragmentation divides a relation "vertically" by columns.
6. A **fragmentation schema** of a database is a definition of a set of fragments that includes *all* attributes and tuples in the database.
7. An **allocation schema** describes the allocation of fragments to sites of the DDBS.
8. The main feature that all DDBMS systems have in common is the fact that data and software are distributed over multiple sites connected by some form of communication network.

9.8 Key Words

Distributed Databases, Data Fragmentation, Replication, Distribution, Client Server Architecture

9.9 Self Assessment Questions

1. What are the main reasons for and potential advantages of distributed databases?
2. What are the main software modules of a DDBMS? Discuss the main functions of each of these modules in the context of the client-server architecture.
3. What is a fragment of a relation? What are the main types of fragments? Why is fragmentation a useful concept in distributed database design?
4. Why is data replication useful in DDBMSs? What typical units of data are replicated?
5. What is meant by *data allocation* in distributed database design? What typical units of data are distributed over sites?
6. How is a horizontal partitioning of a relation specified? How can a relation be put back together from a complete horizontal partitioning?
7. How is a vertical partitioning of a relation specified? How can a relation be put back together from a complete vertical partitioning?
8. Discuss what is meant by the following terms: *degree of homogeneity of a* DDBMS, *degree of local autonomy of a* DDBMS, *federated* DBMS,

distribution transparency, fragmentation transparency, replication transparency, multidatabase system.

9. Discuss the naming problem in distributed databases.
10. Discuss the different techniques for executing an equijoin of two files located at different sites. What main factors affect the cost of data transfer?

9.10 References/Suggested Readings

1 Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000

2 Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld

3 Elamasri R . and Navathe, S., Fundamentals of Database Systems (3rd Edition), Pearsson Education, 2000.

MCA 202/MS 11

Author: Abhishek Taneja **Vetter: Dr. Pardeep Bhatia**
Lesson: Conventional Data Models and Systems **Lesson No. : 10**

Structure

10.0 Objectives

At the end of this chapter the reader will be able to:

- Describe the network data modeling concept
- Distinguish between two basic data structures in the network data model: records and sets
- Describe constraints in network data model
- Describe data manipulation in network data model

10.1 Introduction

This chapter provides an overview of the network data model and hierarchical data model. The original network model and language were presented in the CODASYL Data Base Task Group's 1971 report; hence it is sometimes called the DBTG model. Revised reports in 1978 and 1981 incorporated more recent concepts. In this chapter, rather than concentrating on the details of a particular CODASYL report, we present the general concepts behind network-type databases and use the term **network model** rather than CODASYL model or DBTG model.

The original CODASYL/DBTG report used COBOL as the host language. Regardless of the host programming language, the basic database manipulation commands of the network model remain the same. Although the network model and the object-oriented data model are both navigational in nature, the data structuring capability of the network model is much more elaborate and allows for explicit insertion/deletion/modification semantic specification. However, it lacks some of the desirable features of the object models.

There are no original documents that describe the hierarchical model, as there are for the relational and network models. The principles behind the hierarchical model are derived from Information Management System (IMS), which is the dominant hierarchical system in use today by a large number of banks, insurance companies, and hospitals as well as several government agencies.

10.2 Network Data Modeling Concepts

There are two basic data structures in the network model: records and sets.

10.2.1 Records, Record Types, and Data Items

Data is stored in **records;** each record consists of a group of related data values. Records are classified into **record types,** where each record type describes the structure of a group of records that store the same type of information. We give each record type a name, and we also give a name and format (data type) for each **data item** (or attribute) in the record type. Figure 10.1 shows a record type STUDENT with data items NAME, SSN, ADDRESS, MAJORDEPT, and BIRTHDATE.

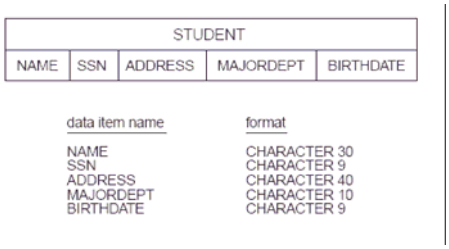


Figure 10.1 A record type STUDENT.

We can declare a virtual data item (or derived attribute) AGE for the record type shown in Figure 10.1 and write a procedure to calculate the value of AGE from the value of the actual data item BIRTHDATE in each record.

A typical database application has numerous record types—from a few to a few hundred. To represent relationships between records, the network model provides the modeling construct called *set type,* which we discuss next.

10.2.2 Set Types and Their Basic Properties

A **set type** is a description of a 1:N relationship between two record types. Figure 10.2 shows how we represent a set type diagrammatically as an arrow. This type of diagrammatic representation is called a **Bachman diagram.** Each set type definition consists of three basic elements:

- A name for the set type.
- An owner record type.
- A member record type.

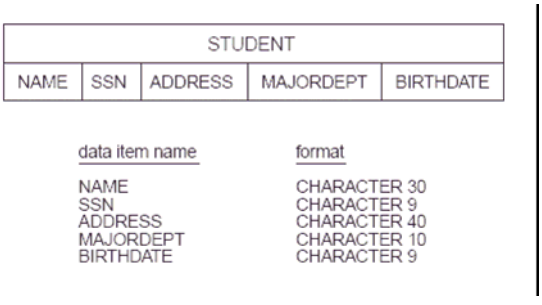


Figure 10.2 The set type MAJOR_DEPT.

The set type in Figure 10.2 is called MAJOR_DEPT; DEPARTMENT is the **owner** record type, and STUDENT is the **member** record type. This represents the 1:N relationship between academic departments and students majoring in those departments. In the database itself, there will be many **set occurrences** (or **set instances**) corresponding to a set type. Each instance relates one record from the owner record type—a DEPARTMENT record in our example—to the set of records from the member record type related to it—the set of STUDENT records for students who major in that department. Hence, each set occurrence is composed of:

- One owner record from the owner record type.
- A number of related member records (zero or more) from the member record type.

A record from the member record type *cannot exist in more than one set occurrence* of a particular set type. This maintains the constraint that a set type represents a 1:N relationship. In our example a STUDENT record can be related to at most one major DEPARTMENT and hence is a member of at most one set occurrence of the MAJOR_DEPT set type.

A set occurrence can be identified either by the *owner record* or by *any of the member records.* Figure 10.3 shows four set occurrences (instances) of the MAJOR_DEPT set type. Notice that each set instance *must* have one owner record but can have any number of member records (**zero** or more). Hence, we usually refer to a set instance by its owner record. The four set instances in Figure 10.3 can be referred to as the 'Computer Science', 'Mathematics', 'Physics', and 'Geology' sets. It is customary to use a different representation of a set instance (Figure 10.4) where the records of the set instance are shown linked together by pointers, which corresponds to a commonly used technique for implementing sets.

In the network model, a set instance is *not identical* to the concept of a set in mathematics. There are two principal differences:

- The set instance has one *distinguished element*—the owner record—whereas in a mathematical set there is no such distinction among the elements of a set.

• In the network model, the member records of a set instance are *ordered*, whereas order of elements is immaterial in a mathematical set. Hence, we can refer to the first, second, i^{th}, and last member records in a set instance. Figure 10.4 shows an alternate "linked" representation of an instance of the set MAJOR_DEPT.

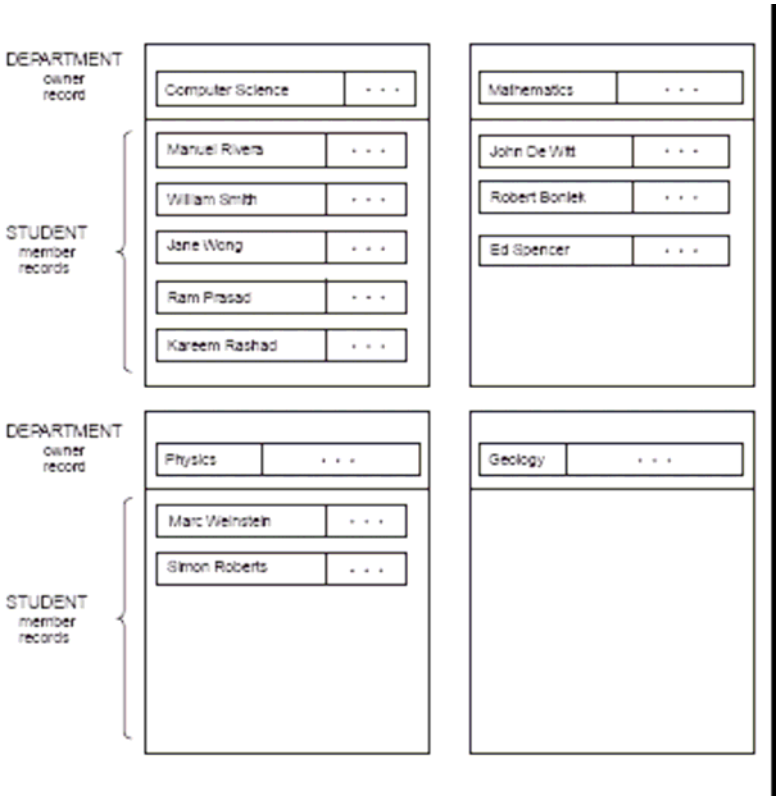


Figure 10.3 Four set instances of the set type MAJOR_DEPT.

In Figure 10.4 the record of 'Manuel Rivera' is the first STUDENT (member) record in the 'Computer Science' set, and that of 'Kareem Rashad' is the last member record. The set of the network model is sometimes referred to as an **owner-coupled set** or **co-set,** to distinguish it from a mathematical set.

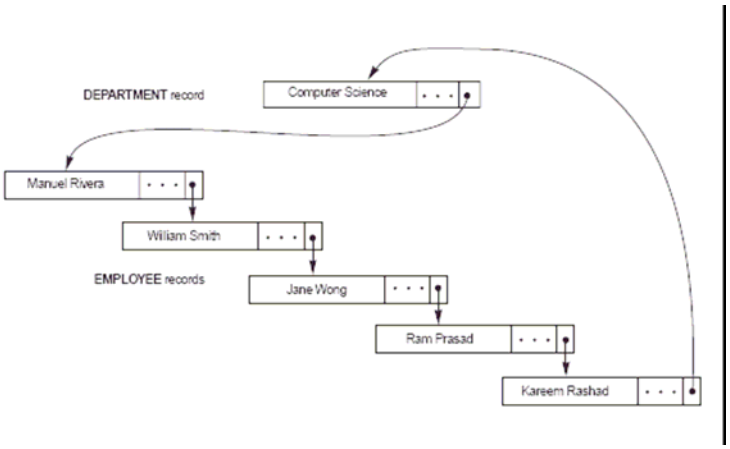


Figure 10.4 Alternate representation of a set instance as a linked list.

10.2.3 Special Types of Sets

System-owned (Singular) Sets

One special type of set in the CODASYL network model is worth mentioning: SYSTEM-owned sets.

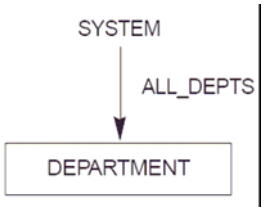


Figure 10.5A singular (SYSTEM-owned) set ALL_DEPTS.

A **system-owned** set is a set with no owner record type; instead, the system is the owner. We can think of the system as a special "virtual" owner record type with only a single record occurrence. System-owned sets serve two main purposes in the network model:

- They provide *entry points* into the database via the records of the specified member record type. Processing can commence by accessing members of that record type, and then retrieving related records via other sets.
- They can be used to *order* the records of a given record type by using the set ordering specifications. By specifying several system-owned sets on the same record type, a user can access its records in different orders.

A system-owned set allows the processing of records of a record type by using the regular set operations. This type of set is called a **singular** set because there is only one set occurrence of it. The diagrammatic representation of the system-owned set ALL_DEPTS is shown in Figure 10.5, which allows DEPARTMENT records to be accessed in order of some field—say, NAME—with an appropriate set-ordering specification. Other special set types include recursive set types, with the same record serving as an owner and a member, which are mostly disallowed; multimember sets containing multiple record types as members in the same set type are allowed in some systems.

10.2.4 Stored Representations of Set Instances

A set instance is commonly represented as a **ring (circular linked list)** linking the owner record and all member records of the set, as shown in Figure 10.4. This is also sometimes called a **circular chain.** The ring representation is symmetric with respect to all records; hence, to distinguish between the owner record and the member records, the DBMS includes a special field, called the **type field,** that has a distinct value (assigned by the DBMS) for each record type. By examining the type field, the system can tell whether the record is the owner of the set instance or is one of the member records. This type field is hidden from the user and is used only by the DBMS.

In addition to the type field, a record type is automatically assigned a **pointer field** by the DBMS for *each set type in which it participates as owner or member.* This pointer can be considered to be *labeled* with the set type name to which it corresponds; hence, the system internally maintains the correspondence between these pointer fields and their set types. A pointer is usually called the **NEXT pointer** in a member record and the **FIRST pointer** in an owner record because these point to the next and first member records, respectively. In our example of Figure 10.4, each student record has a NEXT pointer to the next student record within the set occurrence. The NEXT pointer of the *last member record* in a set occurrence points back to the owner record. If a record of the member record type does not participate in any set instance, its NEXT pointer has a special **nil**

pointer. If a set occurrence has an owner but no member records, the FIRST pointer points right back to the owner record itself or it can be **nil.**

The preceding representation of sets is one method for implementing set instances. In general, a DBMS can implement sets in various ways. However, the chosen representation must allow the DBMS to do all the following operations:

- Given an owner record, find all member records of the set occurrence.
- Given an owner record, find the first, i^{th}, or last member record of the set occurrence. If no such record exists, return an exception code.
- Given a member record, find the next (or previous) member record of the set occurrence. If no such record exists, return an exception code.
- Given a member record, find the owner record of the set occurrence.

The circular linked list representation allows the system to do all of the preceding operations with varying degrees of efficiency. In general, a network database schema has many record types and set types, which means that a record type may participate as owner and member in numerous set types. For example, in the network schema that appears later as Figure 10.8, the EMPLOYEE record type participates as owner in four set TYPES—MANAGES, IS_A_SUPERVISOR, E_WORKSON, and

DEPENDENTS_OF—and participates as member in two set types—WORKS_FOR and SUPERVISEES. In the circular linked list representation, six additional pointer fields are added to the EMPLOYEE record type. However, no confusion arises, because each pointer is labeled by the system and plays the role of FIRST or NEXT pointer for a *specific set type.*

10.2.5 Using Sets to Represent M:N Relationships

A set type represents a 1:N relationship between two record types. This means that *a record of the member record type can appear in only one set occurrence.* This constraint is automatically enforced by the DBMS in the network model. To represent a 1:1 relationship, the extra 1:1 constraint must be imposed by the application program.

An M:N relationship between two record types cannot be represented by a single set type. For example, consider the WORKS_ON relationship between EMPLOYEEs and

PROJECTs. Assume that an employee can be working on several projects simultaneously and that a project typically has several employees working on it. If we try to represent this by a set type, neither the set type in Figure 10.6(a) nor that in Figure 10.6 (b) will represent the relationship correctly. Figure 10.6(a) enforces the incorrect constraint that a PROJECT record is related to only one EMPLOYEE record, whereas Figure 10.6(b) enforces the incorrect constraint that an EMPLOYEE record is related to only one PROJECT record. Using both set types E_P and P_E simultaneously, as in Figure 10.6(c), leads to the problem of enforcing the constraint that P_E and E_P are mutually consistent inverses, plus the problem of dealing with relationship attributes.

The correct method for representing an M:N relationship in the network model is to use two set types and an additional record type, as shown in Figure 10.6(d). This additional record type—WORKS_ON, in our example—is called a **linking** (or **dummy**) record type. Each record of the WORKS_ON record type must be owned by one EMPLOYEE record through the E_W set and by one PROJECT record through the P_W set and serves to relate these two owner records.

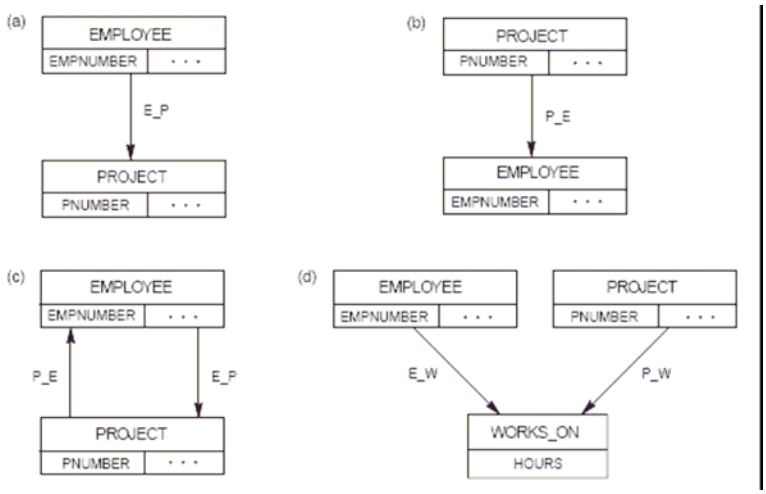


Figure 10.6 Representing M:N relationships. (a)–(c) Incorrect representations. (d) Correct representation using a linking record type.

This is illustrated conceptually in Figure 10.6(e).

Figure 10.6(f) shows an example of individual record and set occurrences in the linked list representation corresponding to the schema in Figure 10.6(d). Each record of the WORKS_ON record type has two NEXT pointers: the one marked NEXT(E_W) points

to the next record in an instance of the E_W set, and the one marked NEXT(P_W) points to the next record in an instance of the P_W set. Each WORKS_ON record relates its two owner records. Each WORKS_ON record also contains the number of hours per week that an employee works on a project. The same occurrences in Figure 10.6(f) are shown in Figure 10.6(e) by displaying the W records individually, without showing the pointers.

To find all projects that a particular employee works on, we start at the EMPLOYEE record and then trace through all WORKS_ON records owned by that EMPLOYEE, using the FIRST(E_W) and NEXT(E_W) pointers. At each WORKS_ON record in the set occurrence, we find its owner PROJECT record by following the NEXT(P_W) pointers until we find a record of type PROJECT. For example, for the E2 EMPLOYEE record, we follow the FIRST(E_W) pointer in E2 leading to W1, the NEXT(E_W) pointer in W1 leading to W2, and the NEXT(E_W) pointer in W2 leading back to E2. Hence, W1 and W2 are identified as the member records in the set occurrence of E_W owned by E2. By following the NEXT(P_W) pointer in W1, we reach P1 as its owner; and by following the NEXT(P_W) pointer in W2 (and through W3 and W4), we reach P2 as its owner. Notice that the existence of direct OWNER pointers for the P_W set in the WORKS_ON records would have simplified the process of identifying the owner PROJECT record of each WORKS_ON record.

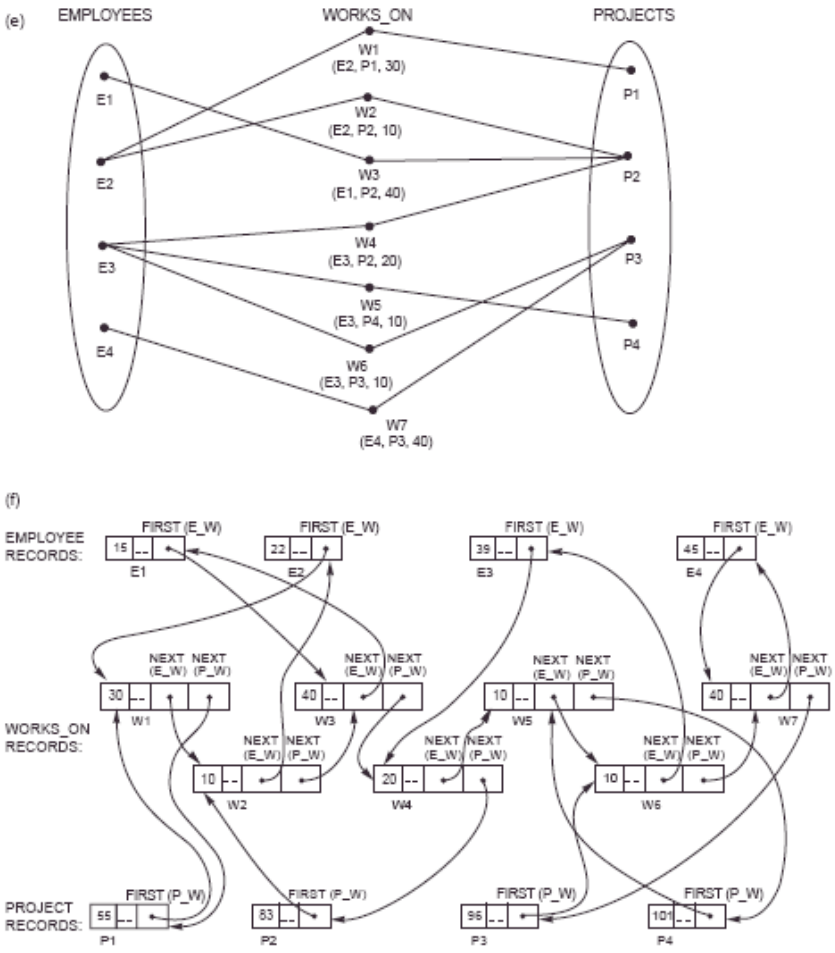


Figure 10.6 *(Continued)* (e) Some instances. (f) Using linked representation.

In a similar fashion, we can find all EMPLOYEE records related to a particular PROJECT. In this case the existence of owner pointers for the E_W set would simplify processing. All this pointer tracing is done *automatically by the DBMS;* the programmer has DML commands for directly finding the owner or the next member.

Notice that we could represent the M:N relationship as in Figure 10.6(a) or Figure 10.6(b) if we were allowed to duplicate PROJECT (or EMPLOYEE) records. In Figure 10.6(a) a PROJECT record would be duplicated as many times as there were employees working on the project. However, duplicating records creates problems in maintaining consistency among the duplicates whenever the database is updated, and it is not recommended in general .

10.3 Constraints in the Network Model

In explaining the network model so far, we have already discussed "structural" constraints that govern how record types and set types are structured. In the present section we discuss "behavioral" constraints that apply to (the behavior of) the members of sets when insertion, deletion, and update operations are performed on sets. Several constraints may be specified on set membership. These are usually divided into two main categories, called **insertion options** and **retention options** in CODASYL terminology. These constraints are determined during database design by knowing how a set is required to behave when member records are inserted or when owner or member records are deleted. The constraints are specified to the DBMS when we declare the database structure, using the data definition language. Not all combinations of the constraints are possible. We first discuss each type of constraint and then give the allowable combinations.

10.3.1 Insertion Options (Constraints) on Sets

The insertion constraints—or options, in CODASYL terminology—on set membership specify what is to happen when we insert a new record in the database that is of a member record type. A record is inserted by using the STORE command. There are two options:

- AUTOMATIC: The new member record is *automatically connected* to an appropriate set occurrence when the record is inserted.
- MANUAL: The new record is not connected to any set occurrence. If desired, the programmer can explicitly (*manually*) connect the record to a set occurrence subsequently by using the CONNECT command.

For example, consider the MAJOR_DEPT set type of Figure 10.2. In this situation we can have a STUDENT record that is not related to any department through the MAJOR_DEPT set (if the corresponding student has not declared a major). We should therefore declare the MANUAL insertion option, meaning that when a member STUDENT record is inserted in the database it is not automatically related to a DEPARTMENT record through the MAJOR_DEPT set. The database user may later insert the record "manually" into a set instance when the corresponding student declares a

major department. This manual insertion is accomplished by using an update operation called CONNECT, submitted to the database system.

The AUTOMATIC option for set insertion is used in situations where we want to insert a member record into a set instance automatically upon storage of that record in the database. We must specify a criterion for *designating the set instance* of which each new record becomes a member. As an example, consider the set type shown in Figure 10.7(a), which relates each employee to the set of dependents of that employee. We can declare the EMP_DEPENDENTS set type to be AUTOMATIC, with the condition that a new DEPENDENT record with a particular EMPSSN value is inserted into the set instance owned by the EMPLOYEE record with the same SSN value.

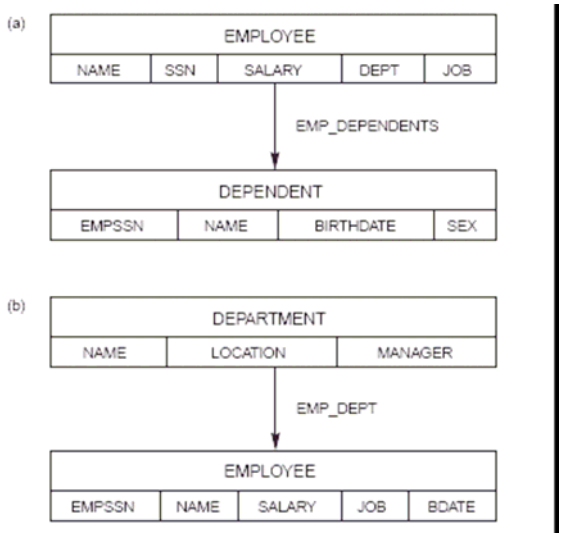


Figure 10.7 Different set options. (a) An AUTOMATIC FIXED set. (b) An AUTOMATIC MANDATORY set.

10.3.2 Retention Options (Constraints) on Sets

The retention constraints—or options, in CODASYL terminology—specify whether a record of a member record type can exist in the database on its own or whether it must always be related to an owner as a member of some set instance. There are three retention options:

- OPTIONAL: A member record can exist on its own *without being* a member in any occurrence of the set. It can be connected and disconnected to set

occurrences at will by means of the CONNECT and DISCONNECT commands of the network DML.

- MANDATORY: A member record *cannot* exist on its own; it must *always* be a member in some set occurrence of the set type. It can be reconnected in a single operation from one set occurrence to another by means of the RECONNECT command of the network DML.
- FIXED: As in MANDATORY, a member record *cannot* exist on its own. Moreover, once it is inserted in a set occurrence, it is *fixed;* it *cannot* be reconnected to another set occurrence.

We now illustrate the differences among these options by examples showing when each option should be used. First, consider the MAJOR_DEPT set type of Figure 10.2. To provide for the situation where we may have a STUDENT record that is not related to any department through the MAJOR_DEPT set, we declare the set to be OPTIONAL. In Figure 10.7(a) EMP_DEPENDENTS is an example of a FIXED set type, because we do not expect a dependent to be moved from one employee to another. In addition, every DEPENDENT record must be related to some EMPLOYEE record at all times. In Figure 10.7(b) a MANDATORY set EMP_DEPT relates an employee to the department the employee works for. Here, every employee must be assigned to exactly one department at all times; however, an employee can be reassigned from one department to another.

By using an appropriate insertion/retention option, the DBA is able to specify the behavior of a set type as a constraint, which is then *automatically* held good by the system.

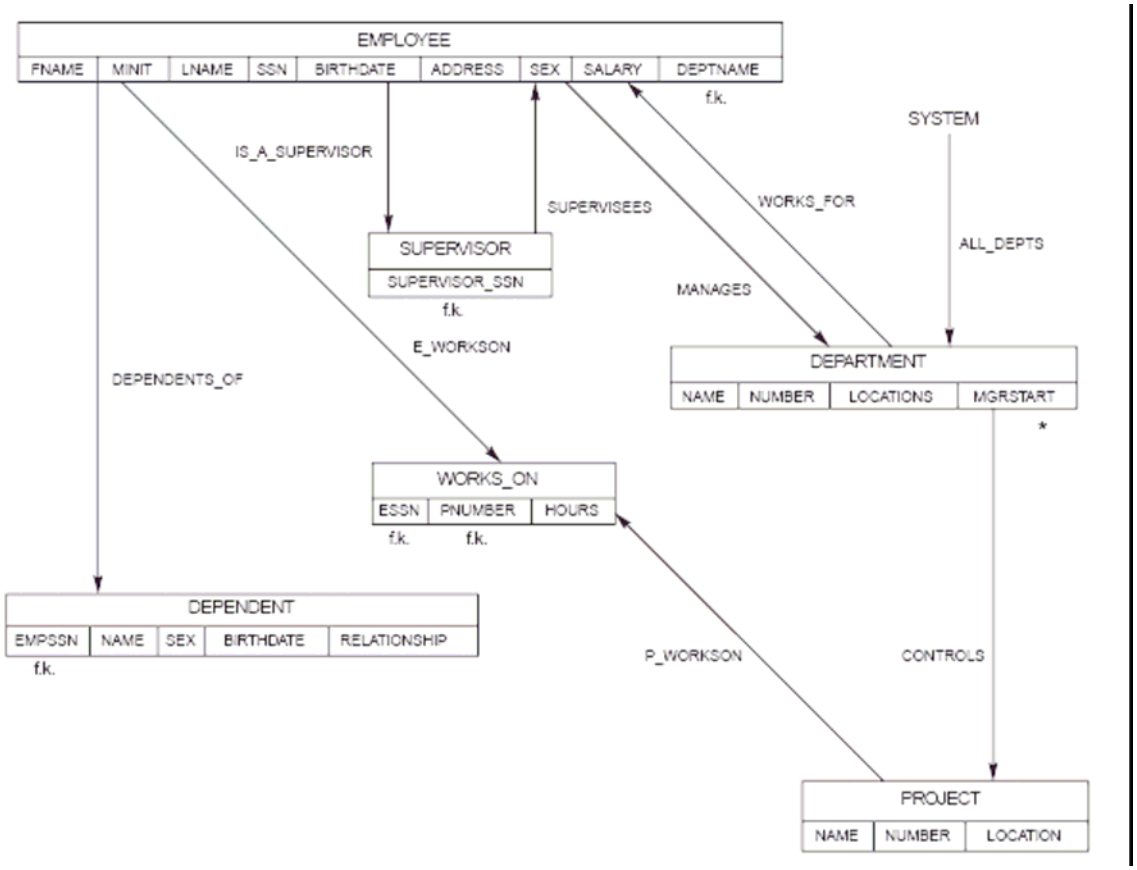


Figure 10.8 A network schema diagram for the COMPANY database.

10.4 Data Manipulation in a Network Database

In this section we discuss how to write programs that manipulate a network database—including such tasks as searching for and retrieving records from the database; inserting, deleting, and modifying records; and connecting and disconnecting records from set occurrences. A **data manipulation language (DML)** is used for these purposes. The DML associated with the network model consists of record-at-a-time commands that are embedded in a general-purpose programming language called the **host language.** Embedded commands of the DML are also called the **data sublanguage.** In practice, the most commonly used host languages are COBOL and PL/I. In our examples, however, we show program segments in PASCAL notation augmented with network DML commands.

10.4.1 Basic Concepts for Network Database Manipulation

To write programs for manipulating a network database, we first need to discuss some basic concepts related to how data manipulation programs are written. The database system and the host programming language are two separate software systems that are linked together by a common interface and communicate only through this interface. Because DML commands are record-at-a-time, it is necessary to identify specific records of the database as **current records.** The DBMS itself keeps track of a number of current records and set occurrences by means of a mechanism known as **currency indicators.** In addition, the host programming language needs local program variables to hold the records of different record types so that their contents can be manipulated by the host program. The set of these local variables in the program is usually referred to as the **user work area (UWA).** The UWA is a set of program variables, declared in the host program, to communicate the contents of individual records between the DBMS and the host program. For each record type in the database schema, a corresponding program variable with the same format must be declared in the program.

Currency Indicators

In the network DML, retrievals and updates are handled by moving or **navigating** through the database records; hence, keeping a trace of the search is critical. Currency indicators are a means of keeping track of the most recently accessed records and set occurrences by the DBMS. They play the role of position holders so that we may process new records starting from the ones most recently accessed until we retrieve all the records that contain the information we need. Each currency indicator can be thought of as a record pointer (or record address) that points to a single database record. In a network DBMS, several currency indicators are used:

- *Current of record type:* For each record type, the DBMS keeps track of the most recently accessed record of that record type. If no record has been accessed yet from that record type, the current record is undefined.

- *Current of set type:* For each set type in the schema, the DBMS keeps track of the most recently accessed set occurrence from the set type. The set occurrence is specified by a single record from that set, which is either the owner or one of the

member records. Hence, the current of set (or current set) points to a record, even though it is used to keep track of a set occurrence. If the program has not accessed any record from that set type, the current of set is undefined.

• *Current of run unit (CRU):* A run unit is a database access program that is executing (running) on the computer system. For each run unit, the CRU keeps track of the record most recently accessed by the program; this record can be from *any* record type in the database.

Each time a program executes a DML command, the currency indicators for the record types and set types affected by that command are updated by the DBMS.

Status Indicators

Several **status indicators** return an indication of success or failure after each DML command is executed. The program can check the values of these status indicators and take appropriate action—either to continue execution or to transfer to an error-handling routine. We call the main status variable DB_STATUS and assume that it is implicitly declared in the host program. After each DML command, the value of DB_STATUS indicates whether the command was successful or whether an error or an exception occurred. The most common exception that occurs is the **END_OF_SET (EOS)** exception.

10.5 Hierarchical Database Structures

10.5.1 Parent-Child Relationships and Hierarchical Schemas

The hierarchical model employs two main data structuring concepts: records and parent-child relationships. A **record** is a collection of **field values** that provide information on an entity or a relationship instance. Records of the same type are grouped into **record types.** A record type is given a name, and its structure is defined by a collection of named **fields** or **data items.** Each field has a certain data type, such as integer, real, or string.

A **parent-child relationship type (PCR type)** is a 1:N relationship between two record types. The record type on the 1-side is called the **parent record type,** and the one on the N-side is called the **child record type** of the PCR type. An **occurrence** (or **instance**) of

the PCR type consists of *one record* of the parent record type and a number of records (zero or more) of the child record type.

A **hierarchical database schema** consists of a number of hierarchical schemas. Each **hierarchical schema** (or **hierarchy**) consists of a number of record types and PCR types.

A hierarchical schema is displayed as a **hierarchical diagram,** in which record type names are displayed in rectangular boxes and PCR types are displayed as lines connecting the parent record type to the child record type. Figure 10.9 shows a simple hierarchical diagram for a hierarchical schema with three record types and two PCR types. The record types are DEPARTMENT, EMPLOYEE, and PROJECT. Field names can be displayed under each record type name, as shown in Figure 10.9. In some diagrams, for brevity, we display only the record type names.

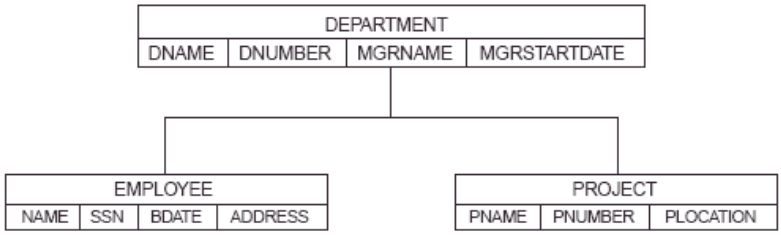


Figure 10.9 A hierarchical schema.

We refer to a PCR type in a hierarchical schema by listing the pair (parent record type, child record type) between parentheses. The two PCR types in Figure 12.1 are (DEPARTMENT, EMPLOYEE) and (DEPARTMENT, PROJECT). Notice that PCR types *do not* have a name in the hierarchical model. In Figure D.01 each *occurrence* of the (DEPARTMENT, EMPLOYEE) PCR type relates one department record to the records of the *many* (zero or more) employees who work in that department. An *occurrence* of the (DEPARTMENT, PROJECT) PCR type relates a department record to the records of projects controlled by that department. Figure 10.10 shows two PCR occurrences (or instances) for each of these two PCR types.

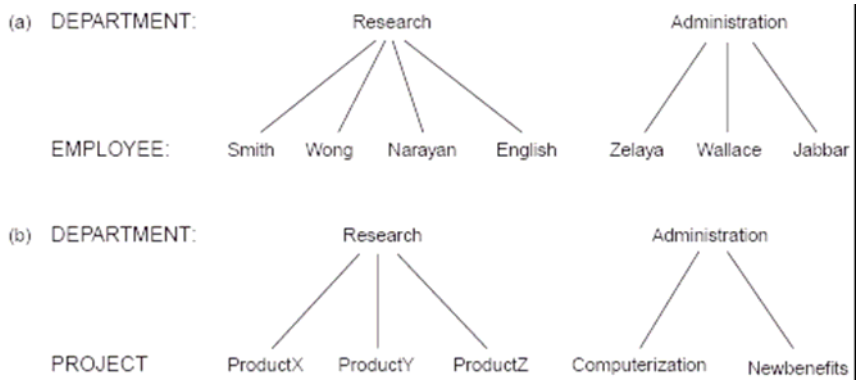


Figure10.10 Occurrences of Parent-Child Relationships.

(a) Two occurrences of the PCR type (DEPARTMENT, EMPLOYEE).

(b) Two occurrences of the PCR type (DEPARTMENT, PROJECT).

10.5.2 Properties of a Hierarchical Schema

A hierarchical schema of record types and PCR types must have the following properties:

1. One record type, called the **root** of the hierarchical schema, does not participate as a child record type in any PCR type.

2. Every record type except the root participates as a child record type in *exactly one* PCR type.

3. A record type can participate as parent record type in any number (zero or more) of PCR types.

4. A record type that does not participate as parent record type in any PCR type is called a **leaf** of the hierarchical schema.

5. If a record type participates as parent in more than one PCR type, then *its child record types are ordered.* The order is displayed, by convention, from left to right in a hierarchical diagram.

The definition of a hierarchical schema defines a **tree data structure.** In the terminology of tree data structures, a record type corresponds to a **node** of the tree, and a PCR type corresponds to an **edge** (or **arc**) of the tree. We use the terms *node* and *record type,* and *edge* and *PCR type,* interchangeably. The usual convention of displaying a tree is slightly different from that used in hierarchical diagrams, in that each tree edge is shown separately from other edges (Figure 10.11). In hierarchical diagrams the convention is

that all edges emanating from the same parent node are joined together (as in Figure 10.9). We use this latter hierarchical diagram convention.

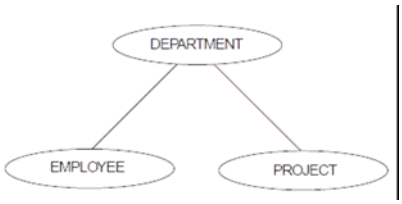


Figure 10.11 A tree representation of the hierarchical schema in Figure 10.9.

The preceding properties of a hierarchical schema mean that every node except the root has exactly one parent node. However, a node can have several child nodes, and in this case they are ordered from left to right. In Figure 10.9 EMPLOYEE is the first child of DEPARTMENT, and PROJECT is the second child. The previously identified properties also limit the types of relationships that can be represented in a hierarchical schema. In particular, M:N relationships between record types *cannot* be directly represented, because parent-child relationships are 1:N relationships, and a record type *cannot participate as child* in two or more distinct parent-child relationships.

An M:N relationship may be handled in the hierarchical model by allowing duplication of child record instances. For example, consider an M:N relationship between EMPLOYEE and PROJECT, where a project can have several employees working on it, and an employee can work on several projects. We can represent the relationship as a (PROJECT, EMPLOYEE) PCR type. In this case a record describing the same employee can be duplicated by appearing once under *each* project that the employee works for. Alternatively, we can represent the relationship as an (EMPLOYEE, PROJECT) PCR type, in which case project records may be duplicated

Example Consider the following instances of the EMPLOYEE:PROJECT relationship:

Project	Employees Working on the Project
A	E1, E3, E5
B	E2, E4, E6
C	E1, E4
D	E2, E3, E4, E5

If these instances are stored using the hierarchical schema (PROJECT, EMPLOYEE) (with PROJECT as the parent), there will be four occurrences of the (PROJECT, EMPLOYEE) PCR type—one for each project. The employee records for E1, E2, E3, and E5 will appear *twice each* as child records, however, because each of these employees works on two projects. The employee record for E4 will appear three times—once under each of projects B, C, and D and may have number of hours that E4 works on each project in the corresponding instance.

To avoid such duplication, a technique is used whereby several hierarchical schemas can be specified in the same hierarchical database schema. Relationships like the preceding PCR type can now be defined across different hierarchical schemas. This technique, called **virtual relationships,** causes a departure from the "strict" hierarchical model. We discuss this technique in next section.

10.6 Integrity Constraints and Data Definition in the Hierarchical Model

10.6.1 Integrity Constraints in the Hierarchical Model

A number of built-in **inherent constraints** exist in the hierarchical model whenever we specify a hierarchical schema. These include the following constraints:

1. No record occurrences except root records can exist without being related to a parent record occurrence. This has the following implications:

    a. A child record cannot be inserted unless it is linked to a parent record.

b. A child record may be deleted independently of its parent; however, deletion of a parent record automatically results in deletion of all its child and descendent records.

c. The above rules do not apply to virtual child records and virtual parent records.

2. If a child record has two or more parent records from the *same* record type, the child record must be duplicated once under each parent record.

3. A child record having two or more parent records of *different* record types can do so only by having at most one real parent, with all the others represented as virtual parents. IMS limits the number of virtual parents to one.

4. In IMS, a record type can be the virtual parent in *only one* VPCR type. That is, the number of virtual children can be only one per record type in IMS.

10.7 Summary

1. There are two basic data structures in the network model: records and sets.
2. Data is stored in **records;** each record consists of a group of related data values.
3. Records are classified into **record types,** where each record type describes the structure of a group of records that store the same type of information.
4. To represent relationships between records, the network model provides the modeling construct called *set type.*
5. Each set type definition consists of three basic elements: • A name for the set type. • An owner record type. • A member record type.
6. The constraints in network data model are usually divided into two main categories, called **insertion options** and **retention options** in CODASYL terminology.
7. A **data manipulation language (DML)** is used for inserting, deleting, and modifying records; and connecting and disconnecting records from set occurrences.
8. The hierarchical model employs two main data structuring concepts: records and parent-child relationships.

9. A **record** is a collection of **field values** that provide information on an entity or a relationship instance.
10. A **parent-child relationship type (PCR type)** is a 1:N relationship between two record types.
11. The definition of a hierarchical schema defines a **tree data structure.**
12. To a hierarchical schema, many **hierarchical occurrences,** also called **occurrence trees,** exist in the database. Each one is a **tree structure** whose root is a single record from the root record type.
13. An **occurrence tree** can be defined as the subtree of a record whose type is of the root record type.
14. A **hierarchical database occurrence** as a sequence of all the occurrence trees that are occurrences of a hierarchical schema.
15. A number of built-in **inherent constraints** exist in the hierarchical model whenever we specify a hierarchical schema.

10.8 Key Words

Network Model, Records, Sets, Data Items, Hierarchical Data Model, Parent Child Relationship, Integrity Constraints

10.9 Self Assessment Questions

1. Describe and distinguish the two basic structures of network data model?
2. What are various constraints in network data model? Explain.
3. What do you mean by constraints? How the insertion options constraints are different from retention option constraints
4. Explain the data definition in network data model. Define using examples.
5. How to write programs that manipulate a network database—including such tasks as searching for and retrieving records from the database; inserting, deleting, and modifying records; and connecting and disconnecting records from set occurrences.
6. Write short notes on: data manipulation language, host language and data sublanguage.

7. What are the different data structuring concepts employed by hierarchical model? Explain.
8. Describe and distinguish the record and parent child relationship concepts in hierarchical model?
9. Define various properties of hierarchical schema.
10. Define the following terms :
    a. Occurrence tree
    b. Preorder traversal of tree
    c. Parent and child record type
11. What are the problems while modeling certain type of relationship in hierarchical model? Explain.
12. What are various integrity constraints and data definition in the hierarchical mode?

10.10 References/Suggested Readings

1 Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley, 2000

2 Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECHWorld

3 Elamasri R . and Navathe, S., Fundamentals of Database Systems (3rd Edition), Pearsson Education, 2000.